Workshop Proceedings

ACES^{MB} 2008

First International Workshop on Model Based Architecting and Construction of Embedded Systems

September 29th, 2008, Toulouse, France

Organized in conjunction with MoDELS'08
11th International Conference on Model Driven Engineering Languages and Systems

Edited by:
Stefan Van Baelen (K.U.Leuven - DistriNet, Belgium)
Iulian Ober (University of Toulouse - IRIT, France)
Susanne Graf (Université Joseph Fourier - CNRS - VERIMAG, France)
Mamoun Filali (University of Toulouse - CNRS - IRIT, France)
Thomas Weigert (Missouri University of Science and Technology, USA)
Sébastien Gérard (CEA - LIST, France)

Table of Contents

Foreword

The development of embedded systems with real-time and other types of critical constraints implies handling very specific architectural choices, as well as various types of critical non-functional constraints (related to real-time deadlines and to platform parameters, such as energy consumption and memory footprint). The last few years have seen a growing interest in (1) using precise (preferably formal) domain-specific models for capturing such dedicated architectural and non-functional information, and (2) using model-driven engineering (MDE) techniques for combining these models with platform independent functional models to obtain a running system. As such, MDE can be used as a means for developing analysis oriented specifications that represent the design model at the same time.

The objective of this workshop is to bring together researchers and practitioners interested in all aspects of model-based software engineering for real-time embedded systems. We target this subject at different levels, from modelling languages and related semantics to concrete application experiments, from model analysis techniques to model-based implementation and deployment. In particular the workshop focus on the following:

- Architecture description languages (ADLs). Architecture models are crucial elements in system and software development, as they capture the earliest decisions that have a huge impact on the realisation of the (non-functional) requirements, the remaining development of the system or software, its deployment, etc. In particular, we are interested in examining:
    - the position of ADLs in an MDE approach
    - the relation between architecture models and other types of models used during requirement engineering (e.g., SysML), design (e.g., UML), etc.
    - techniques for deriving architecture models from requirements, and deriving high-level design models from architecture models
    - verification and early validation using architecture models
- Domain specific design and implementation languages. To achieve the high confidence levels required from critical embedded systems through analytical methods, specific languages with particularly well-behaved semantics are often used in practice, such as synchronous languages and models (Lustre/SCADE, Signal/Polychrony, Esterel), time triggered models (TTA, Giotto), scheduling-oriented models (HRT-UML, Ada Ravenscar), etc. We are interested in examining the model-oriented counterparts of such languages, together with the related analysis and development methods.
- Languages for capturing non-functional constraints (UML-MARTE, AADL, OMEGA, etc.)

- Component languages and system description languages (SysML, BIP, FRACTAL, Ptolemy, etc.).

We received 16 submissions from 8 different countries, of which 10 papers were accepted for the workshop. We hope that the contributions for the workshop and the discussions during the workshop will help to contribute and provide interesting new insights in Model Based Architecting and Construction of Embedded Systems.

The ACES^{MB} 2008 organising committee,

Iulian Ober,
Stefan Van Baelen,
Susanne Graf,
Mamoun Filali,
Thomas Weigert,
Sébastien Gérard,

September 2008.

Acknowledgments

The Organising Committee of ACES^{MB} 2008 would like to thank the workshop Program Committee for their helpful reviews.

Nicolas Belloir (LIUPPA, France)
Jean-Michel Bruel (LIUPPA, France)
Agusti Canals (CS, France)
Jean-Marie Farines (UFSC, Brasil)
Peter Feiler (SEI, USA)
Mamoun Filali (CNRS IRIT, France)
Robert France (CSU, USA)
David Garlan (CMU, USA)
Pierre Gaufillet (Airbus, France)
Sébastien Gérard (CEA LIST, France)
Susanne Graf (VERIMAG, France)
Tom Henzinger (EPFL, Switzerland)
Bruce Lewis (US Army, USA)
John Mettenburg (Rockwell Collins, USA)
Alan Moore (The Mathworks, UK)
Iulian Ober (University of Toulouse, France)
Isabelle Perseil (Telecom ParisTech, France)
Dorina Petriu (Carleton University, Canada)
Bernhard Rumpe (TU Braunschweig, Germany)
Douglas C. Schmidt (Vanderbilt University, USA)
Bran Selic (Malina Software, Canada)
Jean-Bernard Stefani (INRIA, France)
Richard Taylor (UCI, USA)
Martin Törngren (KTH Stockholm, Sweden)
Stefan Van Baelen (K.U.Leuven, Belgium)
Tullio Vardanega (University of Padua, Italy)
Eugenio Villar (Universidad de Cantabria, Spain)
François Vernadat (LAAS, France)
Thomas Weigert (Missouri S&T, USA)
Tim Weilkiens (oose GmbH, Germany)
Sergio Yovine (VERIMAG, France)

This workshop is organised as an event in the context of

- The IST-004527 ARTIST2 Network of Excellence on Embedded Systems Design
- The research project EUREKA-ITEA SPICES (Support of Predictable Integration of mission Critical Embedded Systems)

Multi-Level power consumption modelling in the AADL design flow for DSP, GPP, and FPGA

Eric SENN, Johann LAURENT, and Jean-Philippe DIGUET

Université de Bretagne Sud, Lab-STICC,
CNRS UMR3192,
F-56321 LORIENT Cedex, France

Abstract. This paper presents a method that permits to estimate the power consumption of components in the AADL component assembly model, once deployed onto components in the AADL target platform model. This estimation is performed at different levels in the AADL refinement process. Multi-level power models have been specifically developed for the different type of possible hardware targets: General Purpose Processors (GPP), Digital Signal Processors (DSP) and Field Programmable Gate Arrays (FPGA). Three models are presented for a complex DSP (the Texas Instrument C62), a RISC GPP (the PowerPC 405), and a FPGA from Altera (Stratix EP1S80). The accuracy of these models depends on the refinement level. The maximum error introduced ranges from 70% for the FPGA at the first refinement level (only the operating frequency is considered here) to 5% for the GPP at the third refinement level (where the component's actual source code is considered).

1 Introduction

Originally coming from the avionic domain, AADL (*Architecture Analysis & Design Language*) is now commonly used as an input modelling language for real-time embedded systems [1, 2]. It allows the early analysis of the specification, the verification of functional and non functional properties of the system, and even code generation for the targeted hardware platform [3–5]. In the context of the European project SPICES (*Support for Predictable Integration of mission Critical Embedded Systems*) [6], our aim is to enrich the AADL component based design flow to permit energy and power consumption estimations at different levels in the refinement process. However, such early verifications are only possible if power estimations are completed in a reasonable delay. Only at this condition a fast and fruitful exploration of the design space is permitted.

Significant research efforts have been devoted to develop tools for power consumption estimation at different abstraction levels in embedded system design. A lot of those tools however work at the Register Transfer Level (RTL) (this is the case for tools like SPICE, Diesel [7] and Petrol [8]), at the Cycle Accurate Bit Accurate (CABA) level ([9, 10]), and a few tools at the architectural level (Wattch [11] and Simplepower [12]). Such approaches cannot be used at high levels because simulation times at such low abstraction levels become enormous for complete and complex systems, like multiprocessor heterogeneous platforms.

In [13] and [14], the authors present a characterization methodology for generating power models within TLM for peripheral components. The pertinent activities are identified at several levels and granularities. The characterization phase of the activities is performed at the gate level and is used to deduce the power of coarse-grained activities at higher level. Again, applying such approaches for complex processors or complete systems is not doable. Instruction level or functional level approaches have been proposed [15–17]. They however only work at the assembly level, and need to be improved to take into account pipelined architectures, large VLIW instruction sets, and internal data and instruction caches.

We introduced the *Functional Level Power Analysis* (FLPA) methodology which we have applied to the building of high-level power models for different hardware components, from simple RISC processors to complex superscalar VLIW DSP [18, 19], and for different FPGA circuits [20]. In this paper we show how this modelling approach fits into the AADL design flow and how our power models, being interoperable, are used at different refinement levels. Section 2 presents the AADL component based design flow and the deployment of the Platform Independent Models (PIM) to obtain the Platform Specific Model (PSM) of the target. Section 3 presents the methodology for power estimations and the global power analysis of a complete system described with AADL. Section 4 presents the building of power models and define the three refinement levels where they can be used. The power models of the DSP TI C62, the GPP PowerPC 405, and the FPGA Altera Stratix EP1S80 are presented as examples. The accuracy of our power estimations is finally evaluated.

2 AADL design flow

Figure 1 presents the component based AADL design flow. The *AADL component assembly* model contains all the components and connections instances of the application, and references the implementation models of the components instances from the *AADL models library*. The *AADL target platform* model describes the hardware of the physical target platform. This platform is composed of at least one processor, one memory, and one bus entity to home processes and threads execution. The *AADL deployment plan* model describes the AADL-PSM composition process. It defines all the binding properties that are necessary to deploy the processes and services model of the component-based application on the target platform. All those models are combined to obtain the AADL-PSM model of the complete component-based system. The final implementation of the system is obtained afterward through model transformations and code generation.

The *Open Source AADL Environment Tool* (OSATE) [21] permits the specification of a complete system using AADL. It also permits to check some of its functional and non-functional properties. Those verifications rely on the use

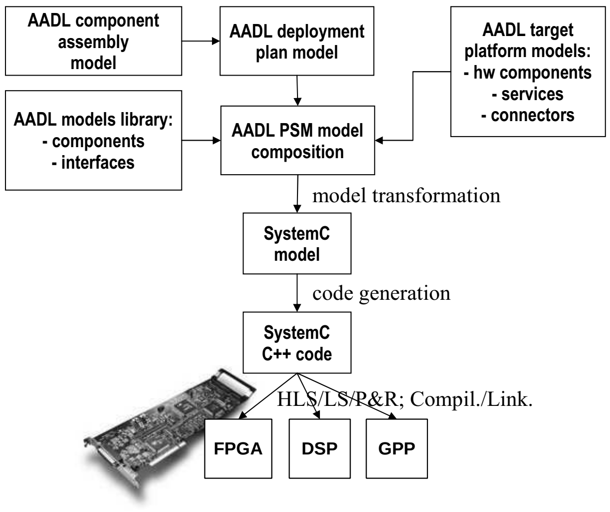

AADL component assembly model
AADL deployment plan model
AADL target platform models: - hw components - services - connectors
AADL models library: - components - interfaces
AADL PSM model composition
model transformation
SystemC model
code generation
SystemC C++ code
HLS/LS/P&R; Compil./Link.
FPGA
DSP
GPP

Fig. 1. AADL component based design flow

of different plug-ins included in the tool set. During the deployment, software components in the component assembly model are bound to hardware components in the target platform model [22]. According to the deployment plan model, OSATE scheduling analysis plug-in uses information embedded in the software components description to propose a binding for the system [23]. Figure 2 shows the typical binding of an application on a multiprocessor architecture. In this example, process `main_process` and its data block `data_b` are bound to the memory `sysRAM`. Threads `control_thread`, `ethernet_driver_thread` and `post_thread` are bound to the first general purpose processor `GPP1`. Thread `pre_thread` is bound to `GPP2`. Thread `hw_thread1` is, like `hw_thread2` a hardware thread. It will be implemented in the reconfigurable FPGA circuit `FPGA1`. One connection between `pre_thread` and `post_thread` has been declared using in and out `data ports` in the threads. This connection is bound to bus `sys_bus` since it connects two threads bound to two different components in the platform. Intra-component connections, like between threads `control_thread` and `ethernet_driver_thread`, do not need to be bound to specific buses. They will however consume hardware resources while being used.

In addition to communication buses, dedicated supply buses can also been declared. A *power analysis* command in the OSATE resources analysis plug-in permits to check if the power capacity of a supply bus is not exceeded. To do that, a power capacity property (`SEI::PowerCapacity`) is declared for a bus, and a power budget is declared for every component that requires an access to this bus (property `SEI::PowerBudget`). The plug-in adds all the power budgets for a bus and compares the result with its power capacity. This mechanism, even if it is interesting, is extremely limited: power budgets for every component are only a guess from the user, and are only used to compute the power consumption

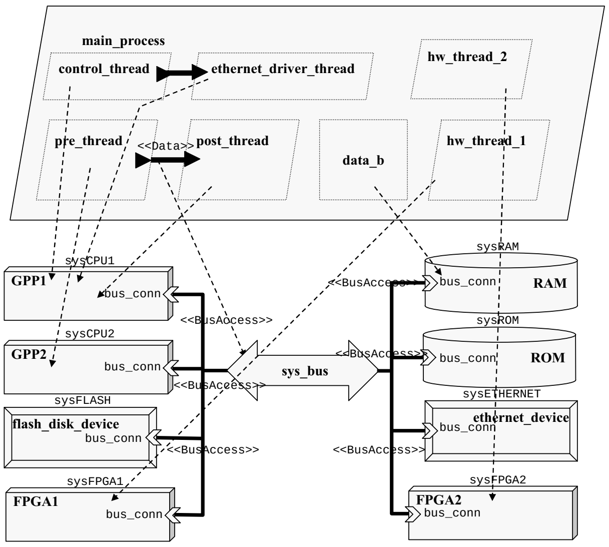


Fig. 2. Binding components to the target platform

of buses in a very simplistic way. In this paper, we propose a method to greatly enhance power analysis in the AADL flow, and to do it in an efficient way not only for buses, but for every consuming component in the system. Moreover, we propose to base power analysis on realistic power estimates, by using an accurate power estimation tool and precise power consumption models for every component in the targeted hardware platform.

3 High-level power consumption estimations

In order to complete power consumption analysis for the whole system, we need first to compute the power budget for every software component in the AADL component assembly model. This is the *power estimation* phase (1) represented with plain edges on figure 3 in the case of the binding of a thread to a processor.

Next, the power budgets of software components are combined to get the power budgets for hardware components. This is the *power analysis* phase (2) represented with thick dotted edges on the figure. Using timing information, the *energy analysis* will be performed afterwards (thin dotted edges).

The challenge for our power estimation tool is to provide a realistic power budget for every component in the application. This tool gathers several information in the system's AADL specification at different places, here from the process and thread descriptions, and from the processor specification. It also uses binding information that may come from a preliminary scheduling analysis. The tool uses the power model of the targeted hardware component (here a processor) to compute the power budget for the (software) component. In fact, it determines the input parameters of the model from the set of information it has gathered. This process is repeated, not only for threads bound to processors, but also for any possible binding of software components onto hardware components, and that means: (i-) threads onto processors or FPGA, (ii-) processes and data onto memories, and (iii-) inter-components connections onto buses.

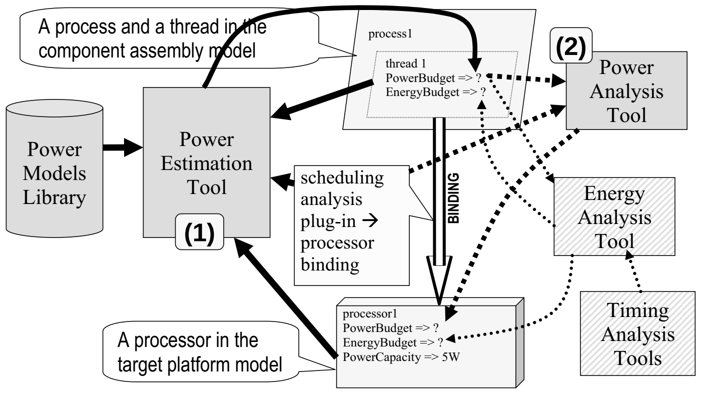


Fig. 3. Power and Energy consumption estimation in the AADL design flow

Once the power budgets have been computed for every component in the application, the power analysis is performed. The power analysis tool retrieves all the component power budgets, together with additional information from the specification, and computes the power budget for every hardware component in the system. Then it computes the power estimation for the whole system. The result of the scheduling analysis (which gives the load of processors) is also taken into account at this level. Indeed, whenever a processor is idle, its power consumption is at the minimum level. Scheduling analysis is performed using basic information on the threads properties defined as properties for each thread implementation in the AADL component assembly model: dispatch_protocol (periodic, aperiodic, sporadic, background), period, deadline, execution_time ...

This paper will concentrate on the power estimation phase, no further details will be given on the power analysis. Energy analysis will be finally performed using information from the timing analysis tools currently being developed by some of our partners in the SPICES project.

4 Multi-level power models

Our Power Estimation Tool, *PET*, is an evolution of the former *SoftExplorer*, initially dedicated to power and energy consumption estimation for processors (from simple RISC General Purpose Processors to very complex VLIW Digital Signal Processors) [24]. This tool comes with a library of power models for every hardware component on the platform.

Our objective is to allow power estimation at different levels in the flow. This involves the use of multi-level power models, which are models that can be used with more or less information, depending on the refinement level. In fact, while the specification is being refined, more information is available and power estimations get more precise.

Let's consider a case study platform including one GPP (the PowerPC 405), one DSP (the Texas Instrument C62), and one FPGA circuit (the Xillinx Virtex 400E). The description of those components' power models can be found respectively in [25], [18], and [20]. Power models are built following our Functional Level Power Analysis methodology [19]. The component's architecture is firstly analysed and relevant parameters regarding its power consumption are identified. Then physical measurements are performed to assess the evolution of the power consumption with the models' input parameters (using little benchmarking programs called "scenario"), and finally power consumption laws are established.

4.1 A complex Digital Signal Processor

The TI C62 processor has a complex architecture. It has a VLIW instructions set, a deep pipeline (up to 15 stages), fixed point operators, and parallelism capabilities (up to 8 operations in parallel). Its internal program memory can be used like a cache in several modes, and an *External Memory Interface* (*EMIF*) is used to load and store data and program from the external memory [26]. In the case of the C62, the 6 following parameters are considered. The *clock frequency* (*F*) and the *memory mode* (*MM*) are what we call *architectural parameters.* They are directly related to the target platform and the hardware component, and can be changed according to the users will. The influence of F is obvious. The C62 maximum frequency is 200MHz (it is for our version of the chip); the designer can tweak this parameter to adjust consumption and performances.

The remaining parameters are called *algorithmic parameters*; they directly depend on the application code itself. The *parallelism rate* α assesses the flow between the processor's instruction fetch stages and its internal program memory controller inside its IMU (Instruction Management Unit). The activity of the processing units is represented by the *processing rate* β. This parameter links the the IMU and the PU (Processing Unit). The activity rate between the IMU and the MMU (Memory Management Unit) is expressed by the *program cache miss rate* γ. The *pipeline stall rate* (*PSR*) counts the number of pipeline stalls during execution. It depends on the mapping of data in memory and on the memory mode.

The memory mode *MM* illustrates the way the internal program memory is used. Four modes are available. All the instructions are in the internal memory in the *mapped mode* (MM_M). They are in the external memory in the *bypass mode* (MM_B). In the *cache mode*, the internal memory is used like a direct mapped cache (MM_C), as well as in the *freeze mode* where no writing in the cache is allowed (MM_F). Internal logic components used to fetch instructions (for instance tag comparison in cache mode) actually depends on the memory mode, and so the power consumption.

A precise description of the C62 power model and its building may be found in [18]. The variation of the power consumption with the input parameters, more precisely the fact that the estimation is not equally sensitive to every parameter, allows to use the model in three different situations.

In the first situation, only the operating frequency is known. The tool returns the average value of the power consumption, which comes from the minimum and maximum values obtained when all the others parameters are being made to vary. The designer can also ask for the maximum value if a higher bound is needed for the power consumption.

In the second situation, we suppose that the architectural parameters (here F and MM) are known. We also assume that the code is not known and that the designer is able to give some realistic values for every algorithmic parameter. If not, default values are proposed, from the values that we have observed running different representative applications on this DSP (see table 1).

In the third situation, the source code is known. It is then parsed by our power estimation tools: the value of every algorithmic parameter is computed and the power consumption is estimated, using the power model and the values enter by the user for the frequency and memory mode.

Table 1. Default algorithmic parameters for the C62

	α	β	PSR
LMSBV_1024	1	0,625	0,385
MPEG_1	0,687	0,435	0,206
MPEG_2_ENC	0,847	0,507	0,28
FFT_1024	0,5	0,39	0,529
DCT	0,503	0,475	0,438
FIR_1024	1	0,875	0,666
EFR_Vocoder_GSM	0,578	0,344	0,045
HISTO (image equalisation by histogram)	0,506	0,346	0,499
SPECTRAL (signal spectral power density estimation)	0,541	0,413	0,288
TREILLIS (Soft Dcision Sequential Decoding)	0,55	0,351	0,038
LPC (Linear Predictive Coding)	0,684	0,468	0,171
ADPCM (Adaptive Differential Pulse Code Modulation)	0,96	0,489	0,194
DCT_2 (imag 128x128)	0,991	0,709	0,435
EDGE DETECTION	0,976	0,838	0,173
G721 (Marcus Lee)	1	0,682	0,032
AVERAGE VALUES	0,7549	0,5298	0,2919

The error introduced by our tool obviously differs in these three situations. To calculate the maximum error, estimations are performed with given values for the parameters known in the situation, and with all the possible values of the remaining unknown parameters. The maximum error comes then from the difference between the average and the maximum estimations. This is repeated for every valid set of known input parameters. The final maximum error is the maximum of the maximum errors. Table 2 gives the maximum error in the three situations above, which correspond to three levels of the specification refinement. Note that the maximal errors computed at level 2 are really pessimistic since we assume here that the designer is completely (100%) wrong on his evaluation of all the input parameters. If his evaluation of those parameters is only 50%, or 25% wrong, then the error introduced by our tool is reduced as well.

Table 2. Maximum errors for the C62 power model (Power in mW)

Known parameters	Memory Mode	Max Power	Min Power	Average Power	Max Error
Level 1					
Frequency	X	3037	848	2076	59%
Level 2					
Frequency, MM, α, β, γ, PSR	Mapped	2954	848	1809	53%
	Cache	2955	756	1778	57%
	Freeze	3018	882	1801	51%
	Bypass	3037	2014	2397	21%
Level 3					
F, MM, and the source code is provided	Max Error = 8%, Average Error = 4%				

4.2 A more simple General Purpose Processor

The PowerPC 405 is a light version of the IBM PowerPC, embedded in the Xilinx VirtexII Pro FPGA. It includes one prefetch instruction unit that allows to reduce the number of pipeline stalls due to instruction misses, two caches (16KB each) (one for the data and the other for instructions). These two caches can be involved separately and use LRU policy. They are two ways associative with 32 bytes line size. And three TLB translating addresses from logical to physical (2 shadow TLB - one for the instructions and one for data - are used and coupled with an unified one).

Measurements show that among the most important factors in the PowerPC 405 consumption are its frequency and the frequency of the bus to which it is connected. The processor can be clocked at 100, 150, 200 or 300 MHz, and, depending on the processor frequency, the bus (OCM or PLB) frequency can take different values between 25 to 100 MHz. Another important parameter to consider is the configuration of the memory hierarchy associated to the processor's core, and that means, which caches are used (data and / or instruction) and where is located the primary memory (internal / external). Once again, the component's power model can be used at three refinement levels.

At the first refinement level, our model gives a rough estimate of the power consumption for the software component, only from the knowledge of the processor and some basic information on its operating conditions. The only information we need is the processor frequency and the frequency of the internal bus (OCM or PLB) to which the processor is connected inside the FPGA. They are two architectural parameters of the PowerPC 405. They will be defined as a property of the AADL processor implementation of the PowerPC 405 in the AADL specification. The maximum error we get here is 27%.

At the second refinement level, we have to add some information about the memories used. We have to indicate which caches will be used in the PowerPC 405 (data cache, instructions cache, or both), and if its primary memory is internal (using the FPGA BRAM memory bank) or external (using a SDRAM accessed through the FPGA I/O). Indeed, while building the power model for the PowerPC 405, we have observed that it draws quite different power and energy in those various situations [25]. Table 3 show the maximal errors we obtain here for every valid set of known input parameters, the others being unknown. The maximum error we obtain is 15,3% and the average error is 6,6%. The first line indicates 0% because in this configuration, there are not remaining unknown parameters that can change the power consumption of the processor.

Table 3. Maximal Errors for the PowerPC 405 at refinement level 2 (Power in mW)

	$F_{processor}/F_{bus}$ (MHz)									
	300/ 100	200/ 100	100/ 100	200/ 66	200/ 50	150/ 50	100/ 50	100/ 33	100/ 25	Max Error
2 caches BRAM	2595	2555	2515	2262	2134	2104	2084	1938	1869	0%
2 caches SDRAM_MAX	3129	3091	3053	2760	2604	2585	2566	2400	2321	
2 caches SDRAM_MIN	2719	2681	2643	2350	2194	2175	2156	1990	1911	
Error	7,0%	7,1%	7,2%	8,0%	8,5%	8,6%	8,7%	9,3%	9,7%	9,7%
BRAM_MAX	2570	2515	2460	2241	2112	2084	2057	1920	1856	
BRAM_MIN	2472	2440	2408	2177	2053	2037	2021	1890	1829	
Error	1,9%	1,5%	1,1%	1,4%	1,4%	1,1%	0,9%	0,8%	0,7%	1,9%
Icache_BRAM&BRAM_MAX	2662	2592	2522	2305	2170	2135	2100	1957	1890	
Icache_BRAM&BRAM_MIN	2497	2451	2405	2193	2071	2048	2025	1897	1836	
Error	3,2%	2,8%	2,4%	2,5%	2,3%	2,1%	1,8%	1,6%	1,4%	3,2%
Icache_SDRAM_MAX	3432	3267	3102	3155	3103	3020	2938	2882	2856	
Icache_SDRAM_MIN	2600	2478	2356	2403	2367	2306	2239	2208	2190	
Error	13,8%	13,7%	13,7%	13,5%	13,5%	13,4%	13,5%	13,2%	13,2%	13,8%
Dcache_BRAM_MAX	2595	2555	2515	2262	2124	2104	2084	1938	1869	
Dcache_BRAM_MIN	2588	2544	2500	2254	2118	2096	2074	1930	1861	
Error	0,1%	0,2%	0,3%	0,2%	0,1%	0,2%	0,2%	0,2%	0,2%	0,3%
Dcache_SDRAM_MAX	3535	3497	3459	3133	2960	2941	2922	2741	2622	
Dcache_SDRAM_MIN	2744	2706	2668	2375	2218	2199	2180	2015	1936	
Error	12,6%	12,8%	12,9%	13,8%	14,3%	14,4%	14,5%	15,3%	15,1%	15,3%

At the lowest refinement level, the actual code of the software component is parsed. In the case of the PowerPC 405, what is important is not exactly what instruction is executed, but rather the type of instruction being executed. We have indeed exhibited that the power consumption changes noticeably from memory access instructions (load or store in memory), to calculation instructions (multiplication or addition). As we have seen before, the place where the data is stored in memory is also important, so the data mapping is also parsed here. The average error we get at this level is 2%. The maximum error is 5%. Logically, that corresponds to the max and average errors for the set of consumption laws for the component.

4.3 Field Programmable Gate Arrays

FPGA (Field Programmable Gate Arrays) are now very common in electronic systems. They are often used in addition to GPP (General Purpose Processors) and / or DSP (Digital Signal Processors) to tackle data intensive dedicated parts of an application. They act as hardware accelerators where and when the application is very demanding regarding the performances, that typically for signal or image processing algorithms. In this case again power estimation can be performed at different refinement levels.

At the highest levels, the code of the application is not known yet. The designer needs however to quickly evaluate the application against power, energy and / or thermal constraints. A fast estimation is necessary here, and a much larger error is acceptable. The parameters we can use from the high-level specifications are the frequency F and the occupation ratio β of the targeted FPGA implementation, that we consider as architectural parameters, and the activity rate α. The experienced designer is indeed able to provide, even at this very high-level, a realistic guess of those parameters' value. As explained before, to obtain the model, i.e. the mathematical equation linking its output to the parameters, we performed a set of different measurements on the targeted FPGA. For different values of the occupation ratio, and for different values of the frequency, we made the activity rate varying and measured the power consumption.

At our first refinement level, only the frequency is known. Our power estimation tool uses the model to estimate, at the given frequency, the power consumption with $\alpha = \beta = 0{,}1$ and with $\alpha = \beta = 0{,}9$. Then it returns the average value between those minimal and maximal values. The maximal errors we obtain for $F = 10\text{MHz}$ and $F = 90\text{MHz}$ (upper bound for the Altera Stratix EP1S80) are given table 4.

At the next refinement level, the two architectural parameters F and β, are known to the user. Like in the case of the former processor's models, default values are proposed for α and also β, coming from a set of representative applications. The maximal error introduced in this case ranges from 6,9% to 44,8%. To determine this error we compute the maximum and minimum estimations for the four extreme (F, β) couples, and compare them to the estimations with α default value.

At the lowest refinement level, the source code (a synthesizable hardware description of the component behaviour, may be written in VHDL or SystemC ...) is used. A High-Level Synthesis tool [27] permits to estimate the amount of resources necessary to implement the application, and given the targeted circuit, to obtain its occupation ratio (β) and its activity rate (α). Those two parameters and the frequency are finally used with the model.

Table 4. Maximum errors for the Altera Stratix EP1S80 (Power in mW)

Known parameters	Max Power	Min Power	Average Power	Max Error
Level 1				
Frequency (F=10MHz)	789	307	548	44%
Frequency (F=90MHz)	4824	835	2830	70%
Level 2				
Frequency, α, β				
F=10MHz, β=0,1	353	307	324	8,8%
F=10MHz, β=0,9	789	544	667	24,1%
F=90MHz, β=0,1	931	835	883	6,9%
F=90MHz, β=0,9	4824	2435	3630	44,8%
Level 3				
F and the source code is provided	Max Error = 4,2%, Average Error = 1,3%			

5 AADL Property Sets

Table 5 and 6 show the property sets associated respectively to the TI C62 and the PowerPC 405 for power estimation at the three refinement levels defined above. Table 7 shows the property set for the FPGA Alter Stratix EP1S80.

Table 5. Property set for the TI C62

TI C62 property set

```
Processor_Frequency : aadlreal applies to (processor);
Processor_Memory_Mode : TIC62::Processor_Memory_Mode_Type applies to (processor);
Processor_Parallelism_Rate : aadlreal applies to (processor);
Processor_Processing_Rate : aadlreal applies to (processor);
Processor_Cache_Miss_Rate : aadlreal applies to (processor);
Processor_Pipeline_Stall_Rate : aadlreal applies to (processor);
Processor_Memory_Mode_Type : type enumeration (CACHE,FREEZE,BYPASS,MAPPED);
Processor_Parallelism_Rate_Default : constant aadlreal => 0,7549;
Processor_Processing_Rate_Default : constant aadlreal => 0,5298;
Processor_Cache_Miss_Rate_Default : constant aadlreal => 0,25;
Processor_Pipeline_Stall_Rate_Default : constant aadlreal => 0,2919;
```

As described section 3, the power estimation tool, when it is invoked, extracts relevant information (a set of parameters) from the AADL specification, then computes the components' power consumption, and returns the results to fill the power budget properties for the software components. The binding makes it possible to put in relation components in the AADL component assembly model with the power models of hardware components on the targeted platform.

As we have just seen, depending on the information refinement, coarse or fine precision power estimations will be performed. Given the refinement level, information to be provided to the estimation tool depends on the selected target (which component). The information is more general if the refinement level is high, it will be more dedicated to the target if the refinement level is low. The set of properties that are used by the estimation tool actually depends on the

Table 6. Property set for the PowerPC 405

PowerPC 405 property set
`Processor_Frequency : aadlreal applies to (processor);`
`Processor_Bus_Frequency : aadlreal applies to (processor);`
`Processor_Primary_Memory : PPC405::Primary_Memory_Type applies to (processor);`
`Processor_Data_Cache : aadlboolean applies to (processor);`
`Processor_Instructions_Cache : aadlboolean applies to (processor);`
`Primary_Memory_Type : type enumeration (BRAM,SDRAM);`

Table 7. Property set for the Altera Stratix EP1S80

FPGA Altera Stratix EP1S80 property set
`FPGA_Frequency : aadlreal applies to (fpga);`
`FPGA_Activity_Rate : TIC62::Processor_Memory_Mode_Type applies to (fpga);`
`FPGA_Occupation_Ratio : aadlreal applies to (fpga);`
`FPGA_Activity_Rate_Default : constant aadlreal => 0,4;`
`FPGA_Occupation_Ratio_Default : constant aadlreal => 0,5;`

component itself, and more precisely, on its power model. Even between two components of the same type, another set of specific properties might be necessary since another set of configuration parameters might apply. This is the case here for the two *processor* components TI C62 and PowerPC405. The property set of the processor comes finally as a part of its power model, and, as this, will remain separated from the general property set associated to the current AADL working project for the application being designed in the OSATE environment.

6 Conclusion

We have presented a method to perform power consumption estimations in the component based AADL design flow. The power consumption of components in the AADL component assembly model is estimated whatever the targeted hardware resource, in the AADL target platform model, is: a DSP (Digital Signal Processor), a GPP (General Purpose Processor), or a FPGA (Field Programmable Gate Array). A power estimation tool has been developed with a library of multi-level power models for those (hardware) components. These models can be used at different levels in the AADL specification refinement process. We have currently defined three refinement levels in the AADL flow. At the lowest level, level 3, the (software) component's actual business code is considered and an accurate estimation is performed. This code, written in C, or C++, for standard threads, can also be written in VHDL or SystemC for hardware threads. At level 2, the power consumption is only estimated from the component operating frequency, and its architectural parameters (mainly linked to its memory configuration in the case of processors). At level 1, the highest level, only the operating frequency of the component is considered.

Three power models have been presented for the TIC62 GPP, the PowerPC405 GPP, and the Altera Stratix EP1S80 FPGA. The maximum errors introduced by these models, at the three refinement levels, are given table 8.

Table 8. Maximal errors summary

Component	Max Error Level 1	Max Error Level 2	Max Error Level 3
TI C62	59%	57%	8%
PowerPC 405	27%	15,3%	5%
Altera Stratix EP1S80	70%	44,8%	4,2%

In the frame of the SPICES project, we are currently working at the integration of our *Power Estimation Tool* and *Power Analysis Tool* in the AADL OSATE tool environment.

References

1. P. Feiler, B. Lewis, and S. Vestal, "The sae architecture analysis & design language (AADL) A standard for engineering performance critical systems," in *IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, october 2006, pp. 1206–1211.
2. "SAE - Society of Automative Engineers, SAES AS5506," v1.0, Embedded Computing Systems Committee, SAE, November 2004.
3. T. Vergnaud, "Modélisation des systèmes temps-réel embarqués pour la génération automatique d applications formellement vérifiées," Ph.D. dissertation, Ecole Nationale Supérieure des Télécommunications de Paris, France, 2006.
4. A. Rugina, K. Kanoun, and M. Kaniche, "Aadl-based dependability modelling," LAAS, Tech. Rep., 2006, number = 06209.
5. J. Hugues, B. Zalila, and L. Pautet, "Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina," in *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07)*. IEEE Computer Society Press, may 2007, pp. 106–112, porto Alegre, Brazil.
6. The SPICES ITEA Project Website. [Online]. Available: http://www.spices-itea.org/
7. P. Research, "Diesel user manual." Philips Electronic Design and Tools Group, Tech. Rep., june 2001.
8. R. Peset-Lopis and K. Goossens, "The petrol approach to high-level power estimation," in *Proceedings of the ISLPED*, Monterey, California, USA, august 1998.
9. M. Loghi, M. Poncino, and L. Benini, "Cycle-accurate power analysis for multiprocessor systems-on-a-chip," in *Proceedings of the GLSVLSI*, Boston, Massachusetts, USA, april 2004.
10. R. BenAtitallah, S.Niar, A.Greiner, S.Meftali, and J.L.Dekeyser, "Estimating energy consumption for an mpsoc architectural exploration," in *in ARCS06*, Frankfurt, Germany, 2006.
11. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. International Symposium on Computer Architecture ISCA'00*, 2000, pp. 83–94.
12. W. Ye, N. Vijaykrishnam, M. Kandemir, and M. Irwin, "The design and use of simplepower: a cycle accurate energy estimation tool," in *Proc. Design Automation Conference DAC'00*, June 2000.
13. N.Dhanwada, I. Lin, and V.Narayanan, "A power estimation methodology for systemc transaction level models," in *International conference on Hardware/software codesign and system synthesis*, 2005.

14. I. Lee, H. Kim, P. Yang, S. Yoo, E. Chung, K.Choi, J.Kong, and S.Eo, "Powervip: Soc power estimation framework at transaction level," in *Proc. ASP-DAC*, 2006.
15. V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *IEEE Trans. VLSI Systems*, vol. 2, pp. 437–445, 1994.
16. S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS'01*, 2001, pp. 3.2.1–3.2.10.
17. G. Qu, N. Kawabe, K. Usami, and M. Potkonjak, "Function-level power estimation methodology for microprocessors," in *Proc. Design Automation Conf. DAC'00*, 2000, pp. 810–813.
18. N. Julien, J. Laurent, E. Senn, and E. Martin, "Power consumption modeling of the TI C6201 and characterization of its architectural complexity," *IEEE Micro, Special Issue on Power- and Complexity-Aware Design*, Sept./Oct. 2003.
19. J. Laurent, N. Julien, E. Senn, and E. Martin, "Functional Level Power Analysis: An efficient approach for modeling the power consumption of complex processors," in *Proc. Design Automation and Test in Europe DATE*, Paris, France, march 2004.
20. E. Senn, N. Julien, N. Abdelli, D. Elleouet, and Y. Savary, "Building and using system, algorithmic, and architectural power and energy models in the fpga design-flow," in *Intl. Conf. on Reconfigurable Communication-centric SoCs 2006*, Montpellier, France, July 2006.
21. The SAE AADL Standard Info Site. [Online]. Available: http://www.aadl.info/
22. H. Balp, E. Borde, G. Hak, and J.-F. Tilman, "Automatic composition of AADL models for the verification of critical component-based embedded systems," in *Proc. of the Thirteenth IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, march 2008, Belfast, Ireland.
23. F. Singhoff, J. Legrand, L. Nana, and L. Marc, "Scheduling and memory requirements analysis with aadl," in *Proceedings of the 2005 annual ACM SIGAda international conference on Ada*, 2005, atlanta, GA, USA.
24. E. Senn, J. Laurent, N. Julien, and E. Martin, "SoftExplorer: Estimating and optimizing the power and energy consumption of a C program for DSP applications," *the EURASIP Journal on Applied Signal Processing, Special Issue on DSP-Enabled Radio*, vol. 2005, no. 16, September 2005.
25. E. Senn, J. Laurent, E. Juin, and J. Diguet, "Refining power consumption estimations in the component based aadl design flow," in *FDL'08, ECSI Forum on specification & Design Languages.*
26. *TMS320C6x User's Guide, Texas Instruments Inc.*, 1999.
27. P. Coussy, G. Corre, P. Bomel, E. Senn, and E. Martin, "High-level synthesis under I/O timing and memory constraints," in *ISCAS'05, International Symposium on Circuits and Systems*, May 2005, kobe, Japan.

VTS-based Specification and Verification of Behavioral Properties of AADL Models[⋆]

D. Monteverde[1,3], A. Olivero[1], S. Yovine[2⋆⋆], and V. Braberman[3⋆⋆⋆]

[1] Inst. de Tecnología, UADE, Argentina. {damonteverde|aolivero}@uade.edu.ar
[2] VERIMAG-CNRS, France. Sergio.Yovine@imag.fr
[3] Departamento de Computación, FCEyN, UBA, Argentina. vbraber@dc.uba.ar

Abstract. AADL is an aerospace standard for model-driven design of complex real-time embedded systems. Currently, behavioral properties of AADL models can be specified inside the system description using AADL concepts or outside it using external textual languages, and verified using schedulability analysis or (Time Petri Net-based) model-checking tools. This paper (1) proposes Visual Timed Scenarios (*VTS*) as a graphical property specification language for AADL, and (2) devises an effective translation from *VTS* to Time Petri Nets (TPN) which enables model-checking properties expressed in *VTS* over AADL models using TPN-based tools integrated into AADL-compliant IDEs (e.g., TOPCASED).

1 Introduction

The Architecture Analysis and Design Language (AADL) [13] is an aerospace standard released by the Society of Automotive Engineers (SAE) for model-based specification and analysis of complex real-time embedded systems. AADL has been designed to support model-based and formal analyses of critical properties. For this, AADL provides modeling concepts for the description of application system architectures in terms of suitable abstractions of software and hardware components and the interactions between them. The definition of AADL motivated the development of AADL-centric tools such as OSATE[4] and Ocarina [10], as well as the integration of AADL into domain-specific model-driven software engineering environments, such as TOPCASED[5]. This enabled different kinds of formal analyses, including schedulability, e.g., with Cheddar [14], and model-checking, e.g., with Time Petri Net-based tools like Tina [4] and Romeo [9].

A way of enhancing the usability of formal techniques in model-driven system design and flows analysis consists in resorting to visual languages capable of representing and visually presenting application semantics in a clear, precise way, specially in the context of event-based systems. Following this idea, in this

⋆ Partially supported by ANPCyT projects PICT 32440 and PICTO-CRUP 31352, STIC-AmSud project TAPIOCA and UBACyT X021
⋆⋆ Currently visiting UBA and UADE.
⋆⋆⋆ Conicet
[4] http://www.aadl.info/
[5] http://www.topcased.org

paper we adopt Visual Timed Scenarios (VTS) [1] as a language for specifying behavioral properties of models of systems described in AADL. In order to make possible the verification of these properties using available tools integrated into AADL-complaint IDEs, we devise a translation from VTS to Time Petri Nets. This allows to use existing model checking tools for verifying if operational descriptions encompassed by AADL model satisfy the translated property. Fig. 1 exhibits the integration of the AADL models with VTS scenarios in a tool chain. The part concerning VTS (enclosed in gray) will be explained in detail along this work.

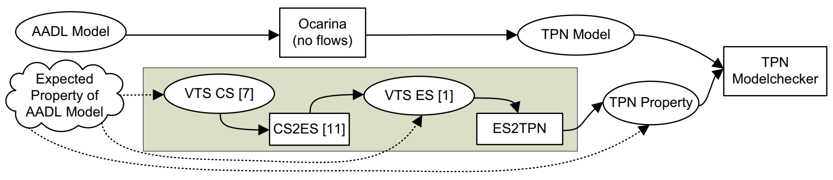


Fig. 1. Tool chain integrating AADL and VTS

The paper is structured as follows. Sec. 2 recalls VTS by means of an example. Sec. 3 briefly reviews Time Petri Nets (TPN). Sec. 4 presents the translation of VTS into TPN. Sec. 5 proposes a procedure to model-check whether a TPN satisfies a property expressed in VTS. Sec. 6 illustrates the application of these results for verifying different behavioral properties of AADL models: (1) mode-change behaviors, and (2) flow specifications.

2 Visual Timed Scenarios (VTS)

2.1 Informal presentation

Visual Timed Scenarios [1, 7] language is used to describe *event patterns*, which can be regarded as simple, graphical depictions of predicates over traces (time-stamped executions), constraining expected behavior. It basically features annotated partial order of relevant events, denoting a (possibly infinite) set of matching traces. Violation of verification goals for models such as freshness, bounded response or event correlation can naturally be expressed using the notation.

The basic elements of VTS graphical notation are points connected by lines and arrows. Points are labeled by sets of events, meaning that the point stands for an occurrence of one of the events in an execution. VTS can represent *precedence relations* and *temporal distances* between points; and sets of events which are *forbidden* between them. The detailed formalization of VTS and its thorough comparison with other visual languages is given in [7]. Here, we informally introduce VTS through a simple, yet illustrative, example.

Consider a system composed of two jobs Job_1 and Job_2 (Fig. 2, based on [2]). The behavior of the system is as follows: (1) Job_1 if started, always terminates; (2) Job_2 if started, always terminates; (3) Job_2 can not start while Job_1 is in execution; (4) Job_1 must terminate in at most 12; (5) Job_2 must wait at least 14 to start; (6) The temporal distance between both jobs' ends is at most 10.

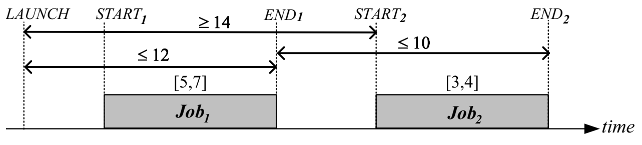


Fig. 2. Example of two jobs

Fig. 3 illustrates these requirements expressed in VTS as *conditional scenarios* [7]. Conditional scenarios allow to state that whenever an *antecedent* sub-scenario (depicted in black) happens, a *consequent* sub-scenario (depicted in gray) must happen too.

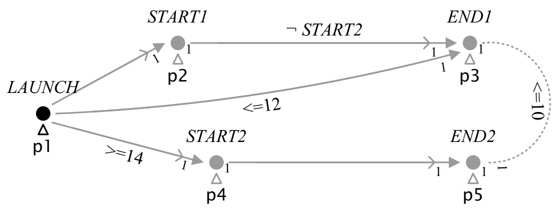


Fig. 3. VTS Conditional scenarios for requirements of two jobs example

Points are labeled with *events*. Triangles below points are used to display optional point names, needed for the formal notation. An arrow between two points specifies a *precedence* relationship. Arrow labels specify *forbidden events* between points: for instance, there is no *START2* event between *START1* and *END1*. A double forward arrow means "the next" occurrence of the event of the destination point (i.e., shorthand for labeling the arrow with the destination's label). A double backward arrow meas "the previous" occurrence of the event of the source point (i.e., shorthand for labeling the arrow with the source's label). A dashed line linking two points expresses a *temporal distance* between them. Dashed lines can also be labeled with forbidden events. Fig. 4 shows the graphical notation of VTS elements used in this work[6].

Verifying conditional scenarios is done by building (using the CS2ES tool showed in Fig. 1) a set of *existential* scenarios that stand for all possible counterexamples of the conditional scenarios [7, 11]. These scenarios, a.k.a. anti-scenarios, model all the ways in which a conditional scenario may be violated by the system. This work only relies on how to model-check existential scenarios, and therefore, hereinafter, existential scenarios are referred as "scenarios". Fig. 5 illustrates all the VTS anti-scenarios of the conditional scenario of Fig. 3.

[6] VTS has more primitives, that increase its expressive power, which are omitted here for the sake of simplicity. The interested reader is referred to [7].

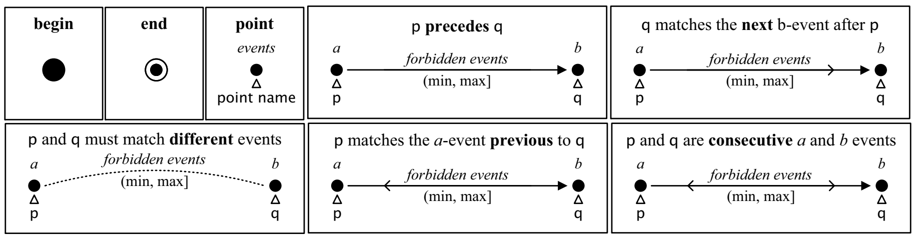
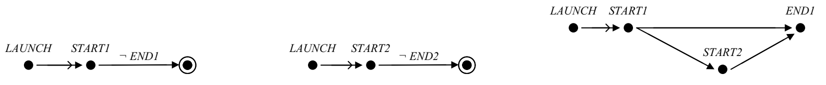
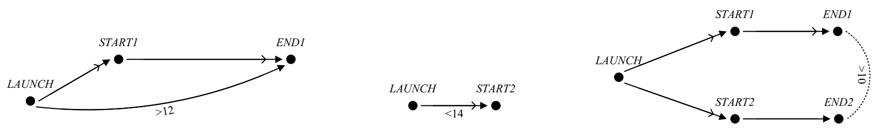

begin
end
point
events
point name
p precedes q
q matches the next b-event after p
p and q must match different events
p matches the a-event previous to q
p and q are consecutive a and b events
a
b
forbidden events
(min, max]
p
q

Fig. 4. *VTS* graphical notation

A big full circle stands for the *begin* of the execution, and two concentric circles correspond to its *end*.



(a) Job_1 starts, but does not terminate (1) (b) Job_2 starts, but does not terminate (2) (c) Job_2 starts while Job_1 is in execution (3)



(d) Job_1 terminates after 12. (4) (e) Job_2 starts before 14. (5) (f) More than 10 passes between jobs ends. (6)

Fig. 5. Anti-scenarios (existential scenarios).

2.2 Formal presentation

Definition 1 (*VTS* syntax). *A* scenario *is a tuple* $\langle \Sigma, P, \ell, \not\equiv, <, \gamma, \delta \rangle$ *, where:*

- Σ *is a finite set of* events*;*
- P *is a finite set of* points*;*
- $\ell : P \to 2^{\Sigma}$ *is labels each point with a non-empty set of events;*
- $\not\equiv \ \subseteq P \times P$ *is an asymmetric relation (*inequality*) between points (graphically represented by* dotted *lines);*
- $< \subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \setminus \{\langle \mathbf{0}, \infty \rangle\}$ *is a* precedence *relation between points (graphically represented by* arrows*), where* $\mathbf{0}$ *and* ∞ *represent the* begin *and the* end *of an execution, resp.;*
- $\gamma : (\not\equiv \cup <) \to 2^{\Sigma}$ *assigns to each pair of points, related by inequality or precedence, the set of events* forbidden *between them;*
- $\delta : (\not\equiv \cup (< \setminus (P \times \{\infty\}))) \to \mathfrak{I}$ *assigns to each inequality or precedence relation an integer-bounded or upper-unbounded interval of non-negative real-numbers restricting the* time elapsed *between the two points.*

Given a set C, a *sequence over* C is a (possibly infinite) sequence of elements from C. Given a sequence s, $|s|$ is its length ($|s| \overset{def}{=} \infty$ when s is infinite) and

$\Pi(s) \overset{def}{=} \{i \in \mathbb{N} \; / \; 0 \leq i < |s|\}$ is the set of positions of s. Given $i, j \in \Pi(s)$, s_i is the i^{th} element of s; $s_{i]}$ is the prefix ending at position i; $s_{[i}$ is the suffix starting at position i and $s_{[i,j]}$ is the sub-sequence from position i to position j (if $i > j$, $s_{[i,j]} \overset{def}{=} s_{[j,i]}$). Using '(' or ')' instead of '[' or ']' means the corresponding sub-sequence does not include its border(s). We call $first(s)$ the first element of s. If s is finite, $last(s)$ is its last element. For $X \subseteq C$, $s \cap X$ denotes the set of elements of X appearing in s, i.e., $\{x \in X \mid \exists i. \; s_i = x\}$.

A *temporal sequence* is a weakly increasing sequence of timestamps (i.e., non negative real numbers). Given a finite temporal sequence τ we define $\Delta(\tau)$ as the time elapsed during τ: $\Delta(\tau) = last(\tau) - first(\tau)$ or 0 if $|\tau| = 0$. A temporal sequence τ can be *shifted* by a real number ϵ producing a temporal sequence called $\tau + \epsilon$, such that $\forall i \in \Pi(\tau); (\tau + \epsilon)_i = \tau_i + \epsilon$.

A *trace* over a set C is a pair $\langle s, \tau \rangle$ where s is a sequence over C and τ is a temporal sequence of the same length. Given a trace $\sigma = \langle s, \tau \rangle$, we define $|\sigma| \overset{def}{=} |s|$ and $\Pi(\sigma) \overset{def}{=} \Pi(s)$. A trace is *time-divergent* iff for any real number T there exists a position k such that $\Delta(\tau_{k]}) > T$.

The semantics of VTS assigns to each scenario a set of traces satisfying it. Labeled points represent events in the traces, they can match a particular position in a trace if the event in that position is among the allowed events associated to the point by the labeling function ℓ.

Intuitively, a *matching* is a mapping between points in a scenario and positions in a trace, exhibiting how the trace satisfies the scenario. Formally:

Definition 2 (VTS semantics). *Given a scenario $\mathcal{S} = \langle \Sigma_{\mathcal{S}}, P, \ell, \not\equiv, <, \gamma, \delta \rangle$, a trace $\sigma = \langle s, \tau \rangle$ over Σ' where $\Sigma_{\mathcal{S}} \subseteq \Sigma'$, and a mapping $\hat{\cdot} : P \to \Pi(\sigma)$; we say that $\hat{\cdot}$ is a* matching *between $\mathcal{S}$ and σ iff for all points $\mathsf{p}, \mathsf{q} \in P$:*

M1 $s_{\hat{\mathsf{p}}} \in \ell(\mathsf{p})$*; the mapping for a point is a position of the trace with an event that labels this point.*

M2 *if* $\mathsf{p} \not\equiv \mathsf{q}$ *then* $\hat{\mathsf{p}} \neq \hat{\mathsf{q}}$*; two different points cannot map to the same position.*

M3 *if* $\mathsf{p} < \mathsf{q}$ *then* $\hat{\mathsf{p}} < \hat{\mathsf{q}}$*; the position of the source point must be smaller than the destination's.*

M4 $s_{(\hat{\mathsf{p}}, \; \hat{\mathsf{q}})} \cap \gamma(\mathsf{p}, \mathsf{q}) = \emptyset$*; no forbidden event can appear in the sub-trace defined by corresponding occurrences of the points.*

M5 $s_{\hat{\mathsf{p}})} \cap \gamma(\mathbf{0}, \mathsf{p}) = s_{(\hat{\mathsf{p}}} \cap \gamma(\mathsf{p}, \infty) = \emptyset$*; no forbidden event specified between begin (resp., a point) and a point (resp., end) can appear before (resp., after) the corresponding occurrence of the point.*

M6 $\Delta(\tau_{[\hat{\mathsf{p}}, \; \hat{\mathsf{q}}]}) \models \delta(\mathsf{p}, \mathsf{q})$*; the time elapsed between the occurrences of the corresponding points must satisfy the specified time restriction.*

M7 $\Delta(\tau_{\hat{\mathsf{p}}]}) \models \delta(\mathbf{0}, \mathsf{p})$*; the time elapsed since begin until the occurrence of a point must satisfy the specified time restriction.*

*Rules **M4-5** and **M6-7** must be considered only when the domains of the functions γ and δ are defined, respectively.*

Definition 3 (Existential VTS Semantics). *We say that a trace σ* satisfies *a scenario $\mathcal{S}$ (noted $\sigma \models \mathcal{S}$) iff there exists at least one matching between them.*

3 Time Petri Nets (*TPNs*)

Time Petri Nets [5] are a widely used formalism for timed systems. They are supported by several tools (e.g. TINA [4], Romeo [9]). TPNs extend Petri nets with temporal intervals associated with transitions: assuming transition t, with an interval $[\alpha, \beta]$, became last enabled at time τ, then t cannot fire earlier than time $\tau + \alpha$ and must fire no later than $\tau + \beta$, unless disabled by firing some other transition. Firing a transition takes no time.

3.1 TPNs Formal Syntax

Definition 4 (Time Petri Net). *A* Time Petri Net[7] *is a tuple* $\mathcal{N} = \langle \boldsymbol{S}, \boldsymbol{T}, \boldsymbol{Pre}, \boldsymbol{Post}, \Sigma_{\mathcal{N}}, \boldsymbol{L}, \boldsymbol{Inh}, \succ, m^0, I^s \rangle$*, where:*

- $\boldsymbol{S}$ *is a finite set of places.*
- $\boldsymbol{T}$ *is a finite set of transitions.*
- $\boldsymbol{Pre} \subseteq \boldsymbol{T} \times \boldsymbol{S}$ *is a relation between transitions and input places.*
- $\boldsymbol{Post} \subseteq \boldsymbol{T} \times \boldsymbol{S}$ *is a relation between transitions and output places.*
- $\Sigma_{\mathcal{N}}$ *is a finite set of events.*
- $\boldsymbol{L} : \boldsymbol{T} \rightarrow \Sigma_{\mathcal{N}} \cup \{\lambda\}$ *is a function that labels each transition with an event or with* $\lambda \notin \Sigma_{\mathcal{N}}$*. We assume that* $\forall\ e \in \Sigma_{\mathcal{N}}, \exists\ t \in \boldsymbol{T},$ *s.t.* $\boldsymbol{L}(t) = e$.
- $\boldsymbol{Inh} \subseteq \boldsymbol{T} \times \boldsymbol{S}$ *is a relation that defines inhibitor places for transitions.*
- $\succ\ \subseteq \boldsymbol{T} \times \boldsymbol{T}$ *is a priority (irreflexive, asymmetric, and transitive) relation between transitions.*
- $m^0 \subseteq \boldsymbol{S}$ *is a set of places with initial marking.*
- $I^s : \boldsymbol{T} \rightarrow \mathfrak{I}$ *is a function called* static interval *function.*

Fig. 6 summarizes the graphical notation for TPNs used in this work.

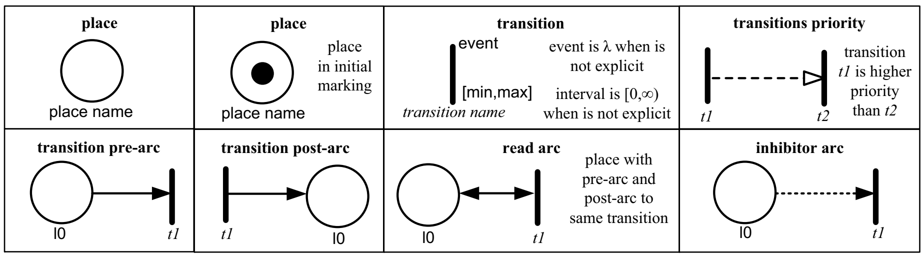


Fig. 6. TPN graphical notation

Parallel Composition. This operation combines two TPNs in one TPN where transitions with the same label (different from λ) are merged. The parallel composition between two TPNs, $\mathcal{N}_1$ and $\mathcal{N}_2$, is denoted as $\mathcal{N}_1 \| \mathcal{N}_2$. See [5] for more details.

[7] For simplicity, we consider here ordinary (i.e. all arcs have weight 1) TPNs, but the results can be extended to non-ordinary ones.

3.2 TPNs Semantics

Given a TPN $\mathcal{N}$, a *state* of $\mathcal{N}$ is a pair $\omega = \langle m, I \rangle$, where $m : \mathbf{S} \to \mathbb{N}$ is a marking and $I : \mathbf{T} \to \mathfrak{I}$ is the interval function that associates a temporal interval with every transition enabled at m. The initial state is denoted ω_0.

The semantics of TPNs defines the evolution of a TPN state resulting from the firing of transitions and passage of time. The reader is referred to [5] for the detailed semantics.

We write $\omega \xrightarrow{\mathbf{L}(t)@\theta} \omega'$ to denote that from state ω, transition t is fired after a time θ, resulting in state ω'; and $\omega \xrightarrow{\lambda@\theta} \omega'$ to denote that from state ω, time can elapse to state ω'. An *execution* is a time-divergent sequence $\rho : \omega_0 \xrightarrow{a_0@\theta_0} \omega_1 \xrightarrow{a_1@\theta_1} \dots$ We write m_{ρ_i} to denote the marking of the i-th state of ρ. The time-divergent trace of ρ is $\sigma = \langle s, \tau \rangle$ with $s = a_0, a_1 \dots$, and $\tau = \vartheta_0, \vartheta_1 \dots$, where $\vartheta_0 = \theta_0$ and $\vartheta_i = \vartheta_{i-1} + \theta_i$, for $i \geq 1$.

4 Translating VTS into TPN

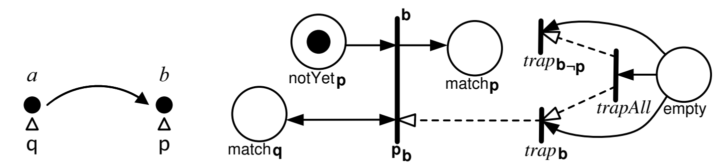
The translation algorithm proceeds as follows: for each part of the VTS scenario that must be matched, it builds a TPN component. So, each point, forbidden event, time restriction, precedence between points, etc., in the VTS scenario, generates a TPN. The translation of the whole scenario is obtained by the *fusion* (a special composition, see below) of all components. The rules that formally define this translation can be found in [12]. The ES2TPN tool (Fig. 1) performs this whole process.

4.1 Construction of TPN components

Construction of TPN components for matching points. In order to recognize occurrences of events as matchings of points, we construct a TPN as follows. For every point p of the VTS scenario, we add two places to the TPN: $\mathsf{notYet_p}$ and $\mathsf{match_p}$. The place $\mathsf{notYet_p}$ has an initial marking and represents that no event labeling point p has occurred yet. The place $\mathsf{match_p}$, if marked, models that a matching event for this point has occurred. Between these places, we add the possible matching transitions: one transition for each event e labeling point p. Each of these is labeled with e, and has a *pre-arc* from $\mathsf{notYet_p}$ and a *post-arc* to $\mathsf{match_p}$. Also, we must consider the case where two (or more) points match the same event. Therefore, we add transitions for all possible combinations of multiple matching points for each event.

To take into account precedence relations among points, for every matching transition into place $\mathsf{match_p}$ we add a *read-arc* from any place $\mathsf{match_q}$, whenever there is a precedence arrow from q to p (this is because place $\mathsf{match_q}$ must be marked before marking place $\mathsf{match_p}$).

Finally, this component has special transitions which will be used in the construction of forthcoming components. Transition $trap_e$ is set with higher priority than any matching transition labeled with event e. Transition *trapAll* has higher priority than all transitions labeled $trap_e$, and therefore higher than all matching transitions (by transitivity). For every point p and event e labeling p, a transition $trap_{e\neg \mathsf{p}}$ is added, with higher priority than any transition matching event e but not matching point p. The purpose of these transitions is to define a priority schema, not to be fired, as they are always disabled by adding a *pre-arc* from a place called empty which is never marked. Fig. 7 gives an example of the construction of TPN component for matching points.

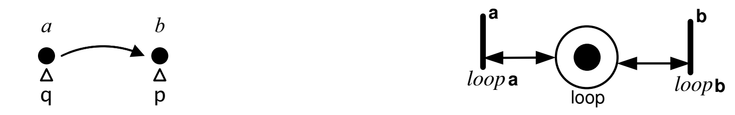


(a) *VTS* scenario (b) TPN component for point p

Fig. 7. TPN component construction for *matching points.*

Construction of TPN components for events not matched by any point. To recognize occurrences of events not associated to point matchings, we add a unique place loop, with an initial marking, and loop transitions for each event e of the scenario.

Fig. 8 shows the resulting TPN for a simple example.



(a) *VTS* scenario (b) TPN of unmatched occurrences of events a and b

Fig. 8. TPN component construction for *unmatched events.*

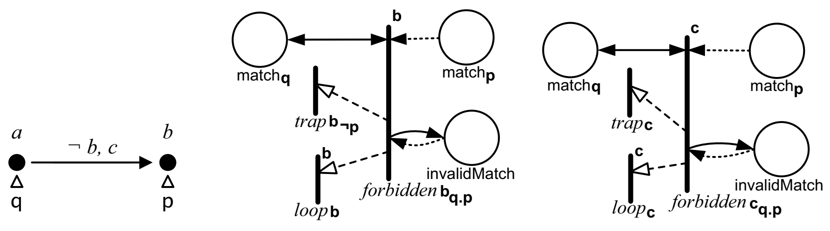
Construction of components for forbidden events on precedence relations. Suppose there is precedence relation from point q to p, and let $\mathsf{match_q}$ and $\mathsf{match_p}$ be the corresponding matching places of the points. For each forbidden event e on the precedence relation, a forbidden transition, labeled with e, is added with a *pre-arc* from $\mathsf{match_q}$ and an *inhibitor-arc* from $\mathsf{match_p}$.

In order for this transition to capture all possible occurrences of the forbidden event e, if e is labeling p, a priority relation is added to transition $trap_e$, otherwise is added to transition $trap_{e\neg \mathsf{p}}$. As we have seen, $trap_e$ has higher priority than any matching transition for event e, and $trap_{e\neg \mathsf{p}}$, has higher priority than any matching transition for event e not related with p.

Also, the corresponding loop transition for event e is disallowed whenever the forbidden transition is enabled, by setting a priority relation. Therefore, the loop transition is enabled only when point q has occurred but not yet point p, avoiding any occurrence of event e not corresponding to the matching point p.

At last, a *post-arc* with an *inhibitor-arc* is added to place invalidMatch. This place, as we will show later, if not empty, avoids reaching the acceptance condition for matching the whole VTS scenario. The purpose of this *inhibitor-arc* is to ensure the boundedness of the TPN.

Fig. 9 illustrates the construction of TPN components for *forbidden events on precedence relations.*



(a) VTS scenario (b) TPN for forb. event b (c) TPN for forb. event c

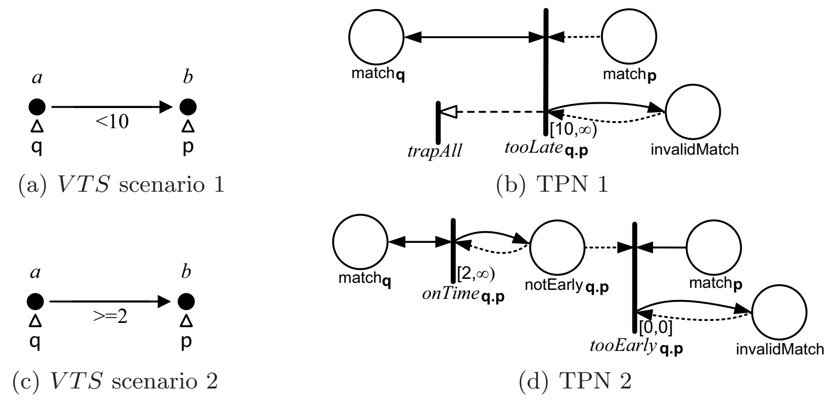
Fig. 9. TPN components for *forbidden events over precedence relations.*

Construction of TPN components for temporal restrictions on precedence relations. In VTS, temporal restrictions over a precedence relation can involve two cases: (1) when the time elapsed between the source and destination points has a maximum allowed value, and (2) when it has a minimum allowed value. Note that VTS time restrictions allow both cases to be combined in an interval constraint.

In case of an upper limit β, we add a transition $tooLate_{\mathsf{q}\cdot\mathsf{p}}$ with a lower bound of β. This transition has a *read-arc* from place $\mathsf{match}_\mathsf{q}$, an *inhibitor-arc* from place $\mathsf{match}_\mathsf{p}$, and a *post-arc* with an *inhibitor-arc* to place invalidMatch. We add a priority relation from this transition to *trapAll* to avoid matching points when it is enabled. Therefore, this transition will avoid point p to match after a time β since point q has occurred. Fig. 10(a) and 10(b) illustrates this construction.

In case of a lower limit α, we use two transitions. One transition, called $onTime_{\mathsf{q}\cdot\mathsf{p}}$, will delay at least α after point q matches, leaving a token at a new place $\mathsf{notEarly}_{\mathsf{q}\cdot\mathsf{p}}$. The other transition, called $tooEarly_{\mathsf{q}\cdot\mathsf{p}}$, has a *pre-arc* from place $\mathsf{match}_\mathsf{p}$, an *inhibitor-arc* from place $\mathsf{notEarly}_{\mathsf{q}\cdot\mathsf{p}}$, and a *post-arc* with an *inhibitor-arc* to place invalidMatch. Therefore, this transition will prevent a scenario matching if point p occurs, but not transition $onTime_{\mathsf{q}\cdot\mathsf{p}}$ which only becomes enabled after a time α since point q's occurrence. Fig. 10(c) and 10(d) illustrates this construction.

Construction of TPN components for restrictions over inequality relations. Consider two points q and p, such that $\mathsf{p} \not\equiv \mathsf{q}$. By definition these points have different matching, then necessarily, either q occurs before p, or p occurs before q. Therefore, both cases must be considered. For this, we apply the rules explained above for taking care of precedence relations.



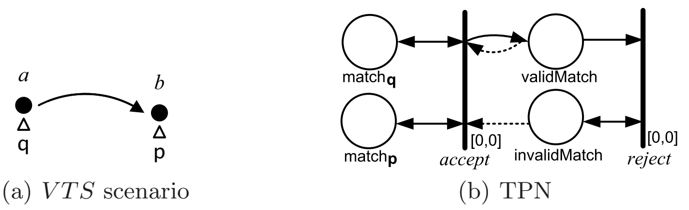
(a) *VTS* scenario 1 (b) TPN 1

(c) *VTS* scenario 2 (d) TPN 2

Fig. 10. TPN components for *time restrictions over precedence relations*

4.2 Construction of the TPN for the whole scenario

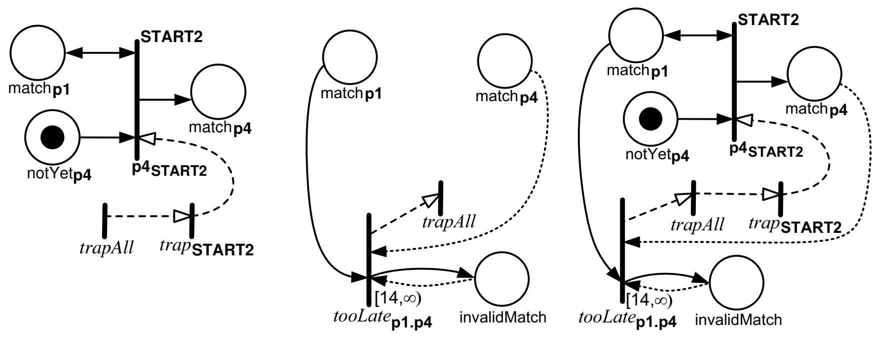
Scenario matching We add a place, namely, validMatch, and two transitions, namely, *accept* and *reject*. Transition *accept*, immediately fires if all points have been matched, and only if place invalidMatch is empty, putting a token in validMatch. Transition *reject*, fires as soon as invalidMatch is reached, removing all tokens (if any) from validMatch. This transition is needed to wait for occurrences of forbidden events in the future. Fig. 11 illustrates this construction.



(a) *VTS* scenario (b) TPN

Fig. 11. TPN component for *scenario matching.*

Fusion of TPNs. Now, we introduce the *fusion* operation, to obtain a TPN by combining two or more TPNs. This operation is based on set union; so if two combined TPNs share places and transitions, these will appear once in the final construction. The fusion operation between two TPNs, $\mathcal{N}_1$ and $\mathcal{N}_2$, is denoted as $\mathcal{N}_1 \oplus \mathcal{N}_2$. Fig. 12 illustrate fusion operation. Resulting fusion of TPNs Fig. 12(a) and Fig. 12(b) is presented in Fig. 12(c).



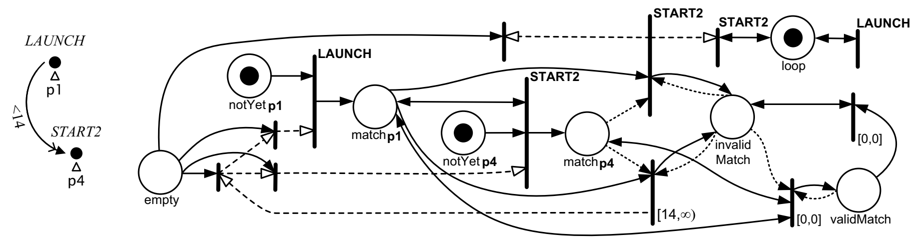
(a) TPN $\mathcal{N}_1$ (b) TPN $\mathcal{N}_2$ (c) TPN $\mathcal{N}_1 \oplus \mathcal{N}_2$

Fig. 12. TPNs' fusion sample

Definition 5. *Given a scenario $\mathcal{S}$, we define the TPN of $\mathcal{S}$, denoted $\mathcal{T}_\mathcal{S}$, as the fusion of the component TPNs constructed as explained above.*

Example. Fig. 13 shows the resulting TPN for the VTS scenario initially presented at Fig. 5(e)[8]. For this scenario, the TPN results from the fusion of the following components:

- **Matching points**: for points p4 (Fig. 12(a)) and p1.
- **Unmatched event**: for events $START2$ and $LAUNCH$.
- **Forbidden events**: for the forbidden event of $START2$ labeling the precedence relation from point p1 to p4.
- **Temporal restrictions**: for time restriction of < 14 labeling the precedence relation from point p1 to p4 (Fig. 12(b)).
- **Scenario Matching**.



(a) VTS (b) TPN

Fig. 13. TPN for scenario: Job_2 starts before 14.

[8] In Fig. 13(b) transition names have been omitted in order to keep the figure small and readable.

5 Model Checking *VTS*

The problem of checking whether a system under analysis (SUA) modeled as a TPN $\mathcal{N}$ satisfies a VTS scenario $\mathcal{S}$ is solved in the following way. The algorithm presented in Sec. 4 translates $\mathcal{S}$ into a TPN (observer) $\mathcal{T}_{\mathcal{S}}$ which recognizes matching traces. $\mathcal{T}_{\mathcal{S}}$ is composed with the SUA $\mathcal{N}$ to check whether a matching execution exists, by using available model checking tools for TPNs. Specifically, the model-checking problem consists in verifying whether there exists an execution that reaches a state where place validMatch of $\mathcal{T}_{\mathcal{S}}$ is marked, and remains marked thereafter.

Property 6. *Given $\mathcal{N}$ and $\mathcal{S}$ then: $\mathcal{N} \| \mathcal{T}_{\mathcal{S}}$ is bounded iff $\mathcal{N}$ is bounded.*

Property 7. *Given $\mathcal{N}$, $\mathcal{S}$ with $\Sigma_{\mathcal{S}} \subseteq \Sigma_{\mathcal{N}}$, and a trace σ over $\Sigma_{\mathcal{N}} \cup \{\lambda\}$ then: σ is a trace of $\mathcal{N} \| \mathcal{T}_{\mathcal{S}}$ iff σ is trace of $\mathcal{N}$.*

Therefore, we are sure that the composition of $\mathcal{N}$ with the TPN $\mathcal{T}_{\mathcal{S}}$ of the scenario preserves the traces of the SUA.

Theorem 8 (Model checking VTS). *Given $\mathcal{N}$ and $\mathcal{S}$ with $\Sigma_{\mathcal{S}} \subseteq \Sigma_{\mathcal{N}}$, then: $\mathcal{N} \vDash \mathcal{S}$ iff there exists a time-divergent execution sequence ρ of $\mathcal{N} \| \mathcal{T}_{\mathcal{S}}$ such that, $\exists k \in \mathbb{N}.\ \forall k' \geq k.\ m_{\rho_{k'}}(\mathsf{validMatch}) = 1$.*

6 Case studies

To carry out our tests, we resort to a *tool chain* that allows us to link the VTS scenarios with AADL models. Based on a property expressed as a VTS conditional scenario, we use the tool presented in [11] to generate the related VTS existential scenarios, that are then translated into TPNs. For this last step, we have developed a tool that implements the translation algorithm described in Sec. 4. On the other hand, the TPN representing the AADL models have been constructed manually[9]. Finally we use the composition of both resulting TPNs as input to the tool Tina, which generates the reachability graph preserving LTL. To check whether the model satisfies the property, we encode Thm. 8 as an LTL model-checking problem and use the `selt` application of the Tina tool-box. For the case studies we analyzed, because `selt` is unable to determine whether an execution is time-divergent, we either relied on the *strongly non-Zeno* [15] hypothesis of the SUA or performed semi-automatic verification. We discuss in the conclusions an approach for automatizing the procedure derived from Thm. 8.

6.1 AADL Mode Change Protocol

In AADL systems, components can operate in different modes, where each of them is associated with a configuration of the component. Changes between modes are triggered by events. A more detailed description can be found in [6].

[9] In the future we plan to use OCARINA [10] or TOPCASED (through FIACRE [3]) to generate them automatically.

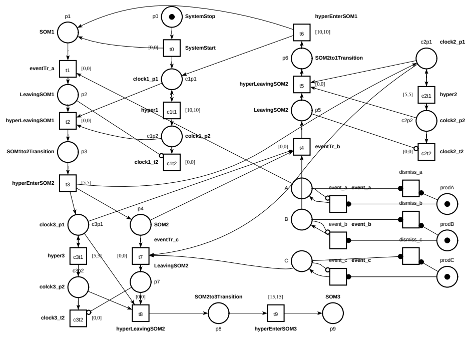


Fig. 14. TPN of the model driver

Fig. 14 shows the TPN of a driver system (extracted from [6]). VTS can be used to analyze and verify different kinds of properties. The mode-change protocol should ensure that the maximum delay between a mode-change request and the entry in the new mode is lower than a specified value. Fig. 15(a) shows a conditional scenario for the verification of this property at the request of $event\_a$. Fig. 15(b) expresses the correlation between the driver events with the environment ones. For example, part of this conditional scenario establishes that if a change to mode SOM2 occurs, a corresponding input event $event\_a$ triggering the mode-change must have occurred. Fig. 15(c) presents a conditional scenario where the antecedent defines an environment behavior by which a certain driver property (the consequent) must be verified. It is important to notice that with VTS we avoid modelling the environment as a (hand-coded) TPN composed with the driver model as proposed at [6], by including its behavior in the scenario as its antecedent. All these scenarios were verified to hold.

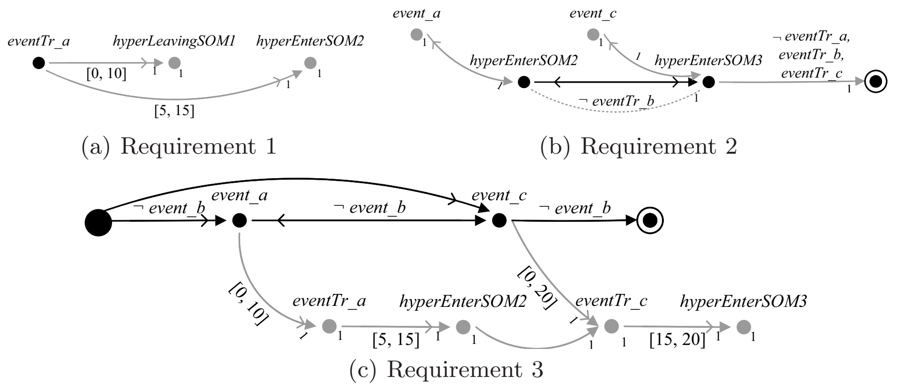
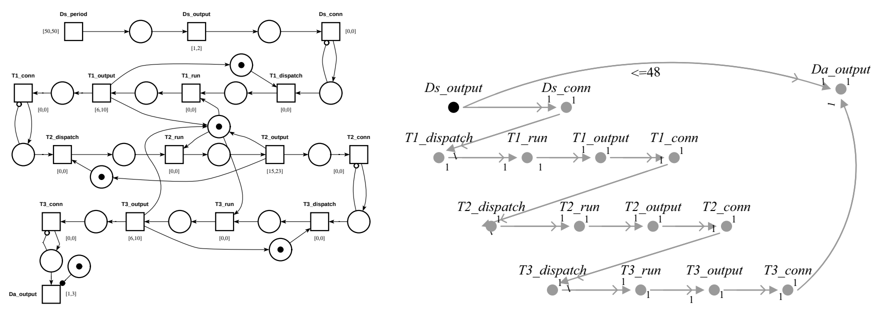


Fig. 15. VTS Conditional Scenarios for verifying Mode-Change example

6.2 AADL Flows Specification

AADL flow specifications are used to describe externally observable sequences of connections through component ports. Flow specifications can be annotated with properties, such as latency, whose verification depend on the properties of the involved components, ports, etc., such as execution times, periods, deadlines, communication delays, etc. Quantitative analysis of flow properties is addressed in [8] and implemented in OSATE. The proposed technique, is based on case-by-case analysis according to the architecture of the sub-components. Here, we propose using VTS scenarios for checking flow latency. We believe the advantages of our approach are twofold. First, it is independent of the architecture of the SUA. Second, it allows specifying non-linear flows, currently not available in AADL. As a case study, we use the example provided in [8]. The TPN of the 3-task system with a periodic sensor and aperiodic tasks and actuator is shown in Fig. 16(a). The VTS scenario for the flow specification is shown in Fig. 16(b). This scenario asserts two properties at once: whenever the sensor produces an output, then (1) the flow is realized, and (2) its latency is less than or equal to 48. Notice that our approach gives a tighter latency than the one in [8].



(a) TPN of aperiodic tasks (b) VTS scenario of flow latency

Fig. 16. Flow latency example (taken from [8])

7 Conclusions and Future Works

This paper proposes an approach for checking complex properties on AADL specifications by relying on the visual language VTS for expressing them. To make it practical, we devised a procedure for generating TPNs from VTS to enable its connection with available IDEs for AADL, such as OSATE and TOPCASED, which integrate TPN-based verification tools.

VTS scenarios proved to be adequate to intuitively express complex properties of AADL models. We also incorporate the idea of using them to describe flows in a more general and independent way. Besides its concrete practical application to AADL-centric system design, the translation presented in this work provides an alternative way to verifying VTS requirements in addition to the one based upon timed automata reachability analysis [1].

Several future research directions are envisaged. First, we plan to generate TOPCASED tool-independent intermediate modeling language FIACRE [3] instead of TPN directly. This will allow model-checking VTS with a larger number of tools integrated by the TOPCASED consortium. Second, we will explore more deeply the connection between VTS and AADL flows. The purpose of this is to investigate whether AADL flow specifications could be extended to cope with non-linear flows. Last but not least, to fully automatize the approach resulting from Thm. 8, a verification procedure which takes into account time-divergence should be implemented for TPNs, adapting, for instance, the algorithms proposed in [15] for timed Büchi automata.

References

1. A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual Timed Event Scenarios. In *Proc. of the 26th ACM/IEEE ICSE '04*. ACM Press, 2004.
2. K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. RTSS '99*. IEEE Comp. Soc. Press, 1999.
3. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: An intermediate language for model verification in the topcased environment. In *4th European Conf. ERTS*, January 2008.
4. B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *3rd Int. Conf. on the Quantitative Evaluation of Systems - (QEST'06)*, 2006.
5. B. Berthomieu and F. Vernadat. State Space Abstractions for Time Petri Nets. In *Handbook of Real-Time and Embedded Systems*, Crc Computer & Information Science Series. Chapman & Hall, July 2007.
6. D. Bertrand, A.-M. Déplanche, S. Faucou, and O. H. Roux. A Study of the AADL Mode Change Protocol. In *13th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE, 2008.
7. V. Braberman, N. Kicillof, and A. Olivero. A Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties. *IEEE Transactions on software Engineering*, 31(12), 2005.
8. P. Feiler and J. Hansson. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Technical Note CMU/SEI-2007-TN-010, Carnegie Mellon University, June 2007.
9. G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A Tool for Analyzing Time Petri Nets. In *Computer Aided Verification*, pages 418–423. Lecture Notes in Computer Science, 2005.
10. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems*, Oct. 2008.
11. D. Monteverde. Verificación Automática de Escenarios Condicionales. Master's thesis, FCEyN. Univ. de Buenos Aires, 2007.
12. D. Monteverde, A. Olivero, S. Yovine, and V. Braberman. VTS-based specification and verification of behavioral properties of AADL models. Technical report, DC. FCEN. UBA. http://www.dc.uba.ar/people/exclusivos/vbraber, 2008.
13. SAE. Architecture Analysis and Design Language. SAE Standard AS5506, 2004.
14. F. Singhoff, A. Plantec, and P. Dissaux. Can We Increase the Usability of Real Time Scheduling Theory? The Cheddar Project. In *Ada-Europe 2008*, 2008.
15. S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed buchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3), May 2005.

Translating AADL into BIP - Application to the Verification of Real-time Systems*

M.Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis

Verimag, Centre Equation - 2, avenue de Vignate 38610 GIERES

Abstract. This paper studies a general methodology and an associated tool for translating AADL (Architecture Analysis and Design Language) and annex behavior specification into the BIP (Behavior Interaction Priority) language. This allows simulation of systems specified in AADL and application to these systems of formal verification techniques developed for BIP, e.g. deadlock detection. We present a concise description of AADL and BIP followed by the presentation of the translation methodology illustrated by a Flight Computer example.

1 Introduction

AADL [5] is used to describe the structure of component-based systems as an assembly of software components mapped onto an execution platform. AADL is used to describe functional interfaces and performance-critical aspects of components. It is used to describe how components interact, and to describe the dynamic behavior of the runtime architecture by providing support for model operational modes and mode transitions. The language is designed to be extensible to accommodate analysis of runtime architectures.

An AADL specification describes the software, hardware, and system part of an embedded real-time system. Basically, an AADL specification is composed of components such as data, subprogram, threads, processes (the software side of a specification), processors, memory, devices and buses (the hardware side of a specification) and system (the system side of a specification).

The AADL specification language is designed to be used with analysis tools that support automatic generation of the source code needed to integrate the system components and build a system executable.

BIP [9] is a language for the description and composition of components as well as associated tools for analyzing models and generating code on a dedicated platform. The language provides a powerful mechanism for structuring interactions involving rendezvous and broadcast.

In order to demonstrate the feasibility of the BIP language and its runtime for the construction of real-time systems, several case studies were carried out such as an MPEG4 encoder [15], TinyOS [10], and DALA [8].

* This work is partially supported by the ITEA/Spices project as well as by the STIC-AmSud project TAPIOCA

This paper provides a general methodology for translating AADL models into BIP models [4]. This allows simulation of systems specified in AADL and application to these systems of formal verification techniques developed for BIP, e.g. deadlock detection [11].

We use existing case studies [3, 2] to validate the methodology. This paper is organized as follows. Section 2 gives an overview of AADL and annex behavior specification. Section 3 gives an overview of BIP. In section 4, we translate AADL components (software, hardware, system and annex behavior specification). We present our tool in Section 5. In section 6, we present a Flight Computer example. Conclusions close the article in Section 7.

2 Overview of AADL

2.1 Generalities

The SAE Architecture Analysis & Design Language (AADL) [5] is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. It plays a central role in several projects such as Topcased [7], OSATE [6], etc.

A system modelled in AADL consists of application software mapped to an execution platform. Data, subprograms, threads, and processes collectively represent application software. They are called *software components*. Processor, memory, bus, and device collectively represent the execution platform. They are called *execution platform components*. Execution platform components support the execution of threads, the storage of data and code, and the communication between threads. Systems are called *compositional components*. They permit software and execution platform components to be organized into hierarchical structures with well-defined interfaces. Operating systems may be represented either as properties of the execution platform or can be modelled as software components.

Components may be hierarchical, i.e. they my contain other components. In fact, an AADL description is almost always hierarchical, with the topmost component being an AADL system that contains, for example, processes and processors, where the processes contain threads and data, and so on.

Compared to other modeling languages, AADL defines low-level abstractions including hardware descriptions. These abstractions are more likely to help design a detailed model close to the final product.

2.2 AADL Components

In this section, we describe the fragment of AADL components, connections and annex behavior taken into account by our translation.

Software Components AADL has the following categories of software components: subprogram, data, thread and process.

Subprogram : A subprogram component represents an execution entry-point in the source text. Subprograms can be called from threads and from other subprograms. These calls are handled sequentially by the threads. A subprogram call sequence is declared in other subprograms or thread implementations.

A subprogram type declaration contains *parameters* (in and out), out *event ports*, and out *event data ports*. A subprogram implementation contains *connections* subclause, a *subprogram calls* subclause, *annex behavior* subclause, and subprogram *property* associations. Figure 1 gives an example of a subprogram, that takes as input two integers A, B, and produces the result as output.

Data : The data component type represents a data type in the source text that defines a representation and interpretation for instances of data. A data implementation can contain *data* subcomponents, and data *property* associations. An example of data is given in Figure 1.

```
subprogram operation
  features
    A: in parameter integer;
    B: in parameter integer;
    result: out parameter integer;
end operation;
```

```
data Person
end Person;
data implementation Person.impl
  subcomponents
    Name : data string;
    Adress: data string;
    Age : data integer;
end Person.impl;
```

Fig. 1. Example of AADL subprogram and data

Thread : A thread represents a sequential flow of control that executes instructions within a binary image produced from source text. A thread always executes within a process. A scheduler manages the execution of a thread.

A thread type declaration contains ports such as *data port*, *event port*, and *event data port*, *subprogram* declarations, and *property* associations. A thread component implementation contains *data* declarations, a *calls* subclause, *annex behavior*, and thread *property* associations.

Threads can have properties. A property has a name, a type and a value. Properties are used to represent attributes and other characteristics, such as the *period*, *dispatch protocol*, and *deadline* of the threads, etc. Dispatch protocol is a property which defines the dispatch behavior for a thread. Four dispatch protocols are supported in AADL: *periodic*, *aperiodic*, *sporadic*, and *background*.

Figure 2 presents a thread component called sensor, that is a periodic thread with inter-arrival time of 20ms. This thread receives an integer data through port inp and sends an event through port outp.

Process : A process represents a virtual address space. Process components are an abstraction of software responsible for executing threads. Processes must contain at least one explicitly declared *thread* or thread group, and can contain a *connections* subclause, and a *properties* subclause. Figure 2 presents an example of process called Partition, that contains thread subcomponents and two types of connections (*data port* and *event port*) between threads.

```
thread sensor
  features
    inp : in data port integer;
    outp : out event port;
  properties
    Dispatch_protocol=>Periodic;
    Period => 20ms;
end sensor;
```

```
process Partition
end Partition;
process implementation Partition.Impl
  subcomponents
      Sensor_A : thread Sensor_Thread.A;
      Data_Fusion: thread Fusion_Thread.Impl;
      Alrm_1 : thread Alrm_Thread.Impl;
  connections
      data port
        Sensor_A.outp->Data_Fusion.inpA;
      event port
        Sensor_A.launch_alrm->Alrm.launch_A;
end Partition.Impl;
```

Fig. 2. Example of AADL thread and process

Hardware Components Execution platform components represent hardware and software that is capable of scheduling threads, interfacing with an external environment, and performing communication for application system connections. We consider two types of hardware components: processors and devices.

Processor : AADL processor components are an abstraction of hardware and software that is responsible for scheduling and executing threads. In other words, a processor may include functionality provided by operating systems.

Device : A device component represents an execution platform component that interfaces with the external environment. A device can interact with application software components through their ports.

Systems A system is the toplevel component of the AADL hierarchy of components. A system component represents a composite component as an assembly of software and execution platform components. All subcomponents of a system are considered to be contained in that system. We present an example of system:

```
system Platform
end Platform;
system implementation Platform.Impl
  subcomponents
      Part : process Partition.Impl;
      p : processor myProcessor ;
```

```
        ...
end Platform.Impl;
```

Annex Behavior Specification Behavior specifications [1] can be attached to AADL model elements using an annex. The behavioral annex describes a transition system attached to subprograms and threads. Behavioral specifications are defined by the following grammar:

```
annex behavior_specification {**
        <state variables>? <initialization>? <states>? <transitions>?
**};
```

- *State variables* section declares typed identifiers. They must be initialized in the *initialization* section.
- *States* section declares automaton states.
- *Transitions* section defines transitions from a source state to a destination state. The transition can be guarded with events or boolean conditions. An action part can be attached to a transition.

Connections A *connection* is a linkage that represents communication of data and control between components. This can be the transmission of control and data between ports of different threads or between threads and processor or device components. There are two types of connections: port connections, and parameter connections.

Port connection: Port connections represent transfer of data and control between two concurrently executing components. There are three types of port connections: *event*, *data* and *event data*.

Parameter connection: represent flow of data between the parameters of a sequence of subprogram calls in a thread.

3 The BIP component framework

BIP (Behavior Interaction Priority) is a framework for modeling heterogeneous real-time components [9]. The BIP component model is the superposition of three layers: the lower layer describes the behavior of a component as a set of transitions (i.e. a finite state automaton extended with data); the intermediate layer includes connectors describing the interactions between transitions of the layer underneath; the upper layer consists of a set of priority rules used to describe scheduling policies for interactions. Such a layering offers a clear separation between component behavior and structure of a system (interactions and priorities).

The BIP framework consists of a language and a toolset including a frontend for editing and parsing BIP programs and a dedicated platform for model validation. The platform consists of an Engine and software infrastructure for executing models. It allows state space exploration and provides access to model-checking tools of the IF toolset [13] such as Aldebaran [12], as well as the D-Finder tool [11]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants [11] and schedulability. The BIP language allows hierarchical construction [14] of composite components from atomic ones by using connectors and priorities.

An *atomic* component consists of a set of *ports* used for the synchronization with other components, a set of transitions and a set of local variables. Transitions describe the behavior of the component. They are represented as a labeled relation between *control states*. A transition is labeled with a port p, a guard g and a function f written in C. The guard g is a boolean expression on local variables and the function f is a block of C code. When g is true, an interaction involving p may occur, in which case f is executed. The interactions between components are specified by connectors.

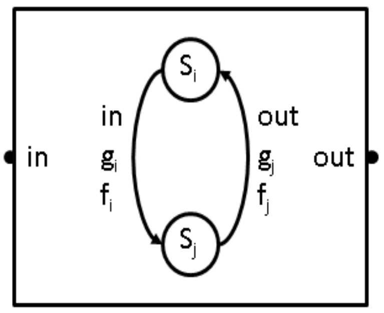


Fig. 3. BIP Atomic Component

Figure 3 shows an atomic component with two control states S_i and S_j, ports *in* and *out*,and corresponding transitions guarded by guard g_i and g_j .

Interactions between components are specified by *connectors*. A connector is a list of ports of atomic components which may interact. To determine the interactions of a connector, its ports have the synchronization attributes *trigger* or *synchron*, represented graphically by a triangle and a bullet, respectively. A connector defines a set of interactions defined by the following rules:

- If all the ports of a connector are synchrons then synchronization is by *rendezvous*. That is, only one interaction is possible, the interaction including all the ports of the connector.
- If a connector has one trigger port then synchronization is by *broadcast*. That is, the trigger port may synchronize with the other ports of the connector. The possible interactions are the non empty sublists containing this trigger port.

In BIP, it is possible to associate with an interaction an activation condition (guard) and a data transfer function both written in C. The interaction is possible if components are ready to communicate through its ports and its activation condition is true. Its execution starts with the computation of data transfer function followed by notification of its completion to the interacting components.

4 Automatic model transformation from AADL to BIP

In this section, we present the translation from AADL [5] to BIP [9]. It is organized in five part. First, we translate AADL software components (subprogram, data, thread and process). Second, we translate hardware components (processor, device). Third, we translate a system component. Fourth, we translate the AADL annex behavior specification [1] in BIP. Finally, we translate connections.

4.1 Software Component

We define the translation of the different AADL software components into BIP.

Subprogram Depending on its structure, we translate the AADL subprograms into atomic or compound BIP components:

As atomic BIP component : When the AADL subprogram does not contain subprogram calls and connections, it is modelled as an atomic component in BIP. Figure 4 shows such a component. This component has two ports *call* and *return*, because subprogram can be called from another subprogram or thread. It also has a particular state *IDLE* and two transitions to express the call and return to the *IDLE* state. The behavior is obtained from the annex as described in section 4.4.

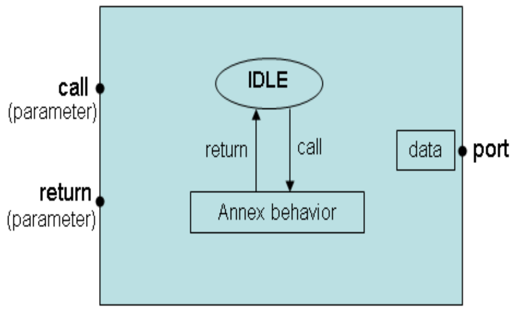


Fig. 4. Subprogram as atomic BIP component

As compound component : When the AADL subprogram contains subprogram calls and connections, it is modelled as a compound BIP component. The subprogram calls are executed sequentially. This execution is modelled by an atomic component with states $wait\_call_1 ... wait\_call_n$ and $wait\_return_1 ... wait\_return_n$, transitions labeled by the ports $call_1 ... call_n$ and $return_1 ... return_n$ (where n is the number of the subprograms called $sub_1 ... sub_n$). To enforce the right sequence of execution and the transfer of parameters, two ports *call* and *return* are used to express calls to the compound subprogram by other subprograms or threads, and the *port data* to sends event or data to the threads, as shown in Figure 5.

Data The data component type represents a data type in the source text that defines a representation and interpretation for instances of data in the source text. In BIP it is transformed into a C data structure.

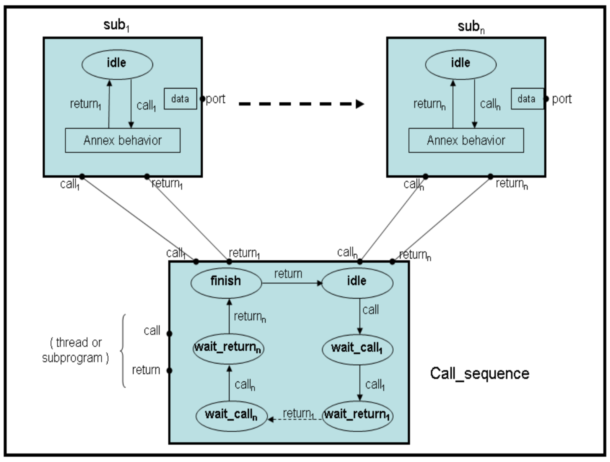


Fig. 5. Subprogram as compound BIP component

Thread : An AADL thread is modelled in BIP by an atomic component as shown in Figure 6. The initial state of the thread is HALTED. On an interaction through port *load* the thread is initialized. Once initialization is completed the thread enters the READY state, if the thread is ready for an interaction through the port *req_exec*. Otherwise, it enters the SUSPENDED state. When the thread is in the SUSPENDED state it cannot be dispatched for execution.

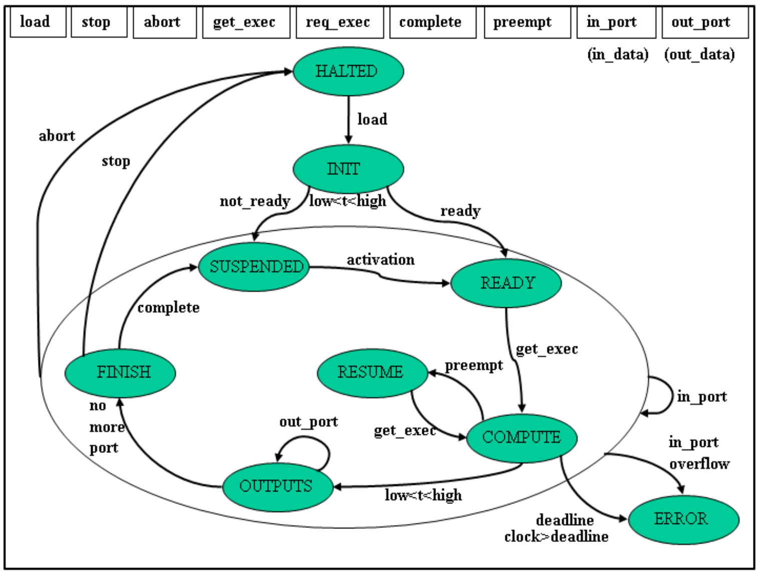


Fig. 6. BIP model for thread behavior

When in the SUSPENDED state, the thread is waiting for an event and/or period to be activated depending on the thread dispatch protocol (periodic, aperiodic, sporadic). In the READY state, a thread is waiting to be dispatched through an interaction in the port *get_exec*. When dispatched, it enters the state COMPUTE to make a computation. Upon successful completion of the computation, the thread goes to the OUTPUTS state. If there are some *out_ports* to dispatch the thread returns to the OUTPUTS state. otherwise, it enters the FINISH state.

The thread may be requested to enter its HALTED state through a port *stop* after completing the execution of a dispatch. A thread may also enter the thread HALTED state immediately through an *abort* port.

Process : Processes must contain at least one explicitly declared thread or thread group. The process behavior is illustrated in Figure 7. Once processors of an execution platform are started, the process enters to the state LOADING through port *load* and it is ready to be loaded.

A process is considered as stopped when all threads of the process are halted. When a process is stopped, each of its threads is given a chance to finalize its execution.

A process can be aborted by using *abort* port. In this case, all contained threads terminate their execution immediately and release all resources.

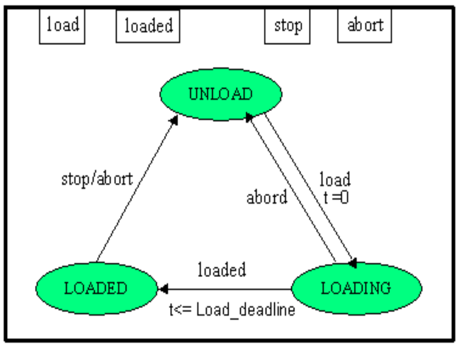


Fig. 7. BIP model for process behavior

The *Load_deadline* property specifies the maximum amount of elapsed time allowed between the time the process begins and completes loading.

4.2 Execution Platform Components

This section defines the translation into BIP of processors and devices.

Processors AADL processor components are an abstraction of hardware and software that is responsible for scheduling and executing threads. Schedulers are modelled as atomic BIP components as shown in Figure 8. The initial state of a scheduler is IDLE. When a thread become ready, the scheduler enters the CHOICE state through an interaction on port *ready*. In this state, the thread ID is stored into the scheduler memory. When a thread is dispatched, the scheduler selects a thread identifier (into *SelectedID* variable) and enters the WAIT_END state through an interaction on port *dispatch*. If there are several threads to be dispatched the scheduler re-enters to the state CHOICE, otherwise, it enters the state IDLE.

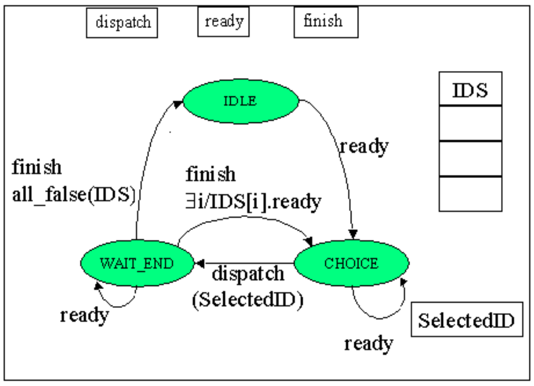


Fig. 8. BIP model of a scheduler

Devices A device component represents an execution platform component that interfaces with the external environment. A device can interact with application software components through their ports. It is modelled as an atomic component in BIP.

4.3 System

A system component represents an assembly of software and execution platform components. All subcomponents of a system are considered to be contained in that system. A system is modelled as a compound component in BIP. Figure 9 shows a BIP component representing a system and connexion between threads, process, and scheduler.

4.4 Annex Behavior specification

Some annex behavior elements can be directly translated to BIP whereas for others we need new BIP facilities. Actual behaviors are supposed to be described using the implementation language. The proposed behavioral annex allows the expression of data dependent behaviors so that more precise behavioral analysis remains possible.

- The *state variables* section declares typed identifiers. In BIP, they correspond to data variables. They must be initialized in the *initialization* section, which is directly included in the BIP initialization part.
- The *states* section declares automaton states as: The *initial* state is directly included in BIP. The *return* state indicates the return to the caller. This case is represented in BIP as a transition from *return* state to *idle* state.
- The *transitions* section defines transitions from a source state to a destination state. Transitions can be guarded with events or boolean conditions, and can contain an action. Each transition is translated as one or a sequence of BIP transitions.

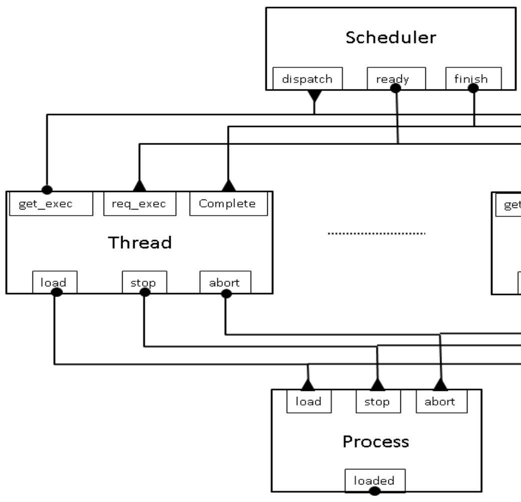


Fig. 9. BIP System

4.5 Connections

Port connection : is translated in BIP depending on the categories :

– an event connection is translated into strong synchronization between the corresponding event ports.
– a data connection is translated into connection with transfer of data.
– an event data connection is translated into a strong synchronization between the corresponding ports with transfer of data.

Parameter connection : is translated in BIP by transfer of data between the parameters of a sequence of subprogram calls in a thread, as shown in section 4.1.

5 Tool

From the high-integrity systems point-of-view, the use of automatic code generation in the development process is profitable. As the generated code is a combination of a relatively small set of extensively tested design patterns, the analysis and review of this code is easier than for hand-written code.

The tool chain is described in Figure 10, and it has the following features:

– *AADL to BIP Transformation:* Using model transformations, allows to perform analysis on the models before code generation. The tool generating BIP from AADL (Figure 10) has been implemented in Java, as a set of plugins for the open source Eclipse platform. It takes an input an AADL model(.aaxl) conforming to the AADL metamodel and generates a BIP model conforming to the BIP metamodel. Models generated may be timed or untimed. Timed

models can be transformed into untimed models in which time progress is represented by a tick port that exists in all timed components and a connector connecting all tick ports.
- *Code Generation:* Takes as input a BIP model and generate the C/C++ code to be executed by the Engine.
- *Exploration Engine:* The engine has a state space exploration mode, which under some restrictions on the data used, generates state graphs that can be analyzed by using finite state model-checking tools.
- *Simulation:* Monitors the state of atomic components and finds all the enabled interactions by evaluating the guards on the connectors. Then, between the enabled interactions, priority rules are used to eliminate the ones with low priority.
- *Verification:* Automatic verification is very useful for early error detection.

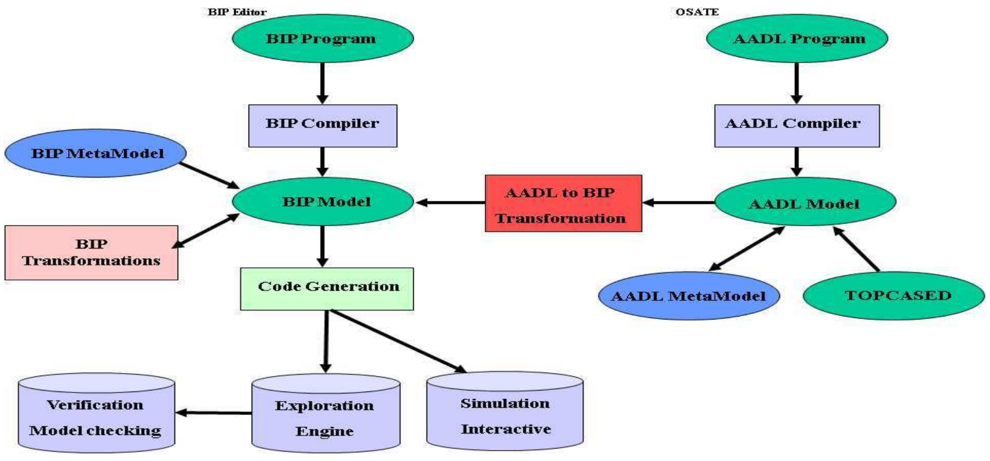


Fig. 10. AADL to BIP Tool Architecture

6 Case studies

We used some examples of AADL [3, 2] (with annex behavior specification) to check the feasibility of our translation from AADL to BIP. In this section, we present the example of a simplistic flight computer [2].

The Flight Computer has a thread called Sensor_Sim that periodically sends integers data for the current AoA(angle-of-attack) and Climb_Rate, and an event in case of Engine_Failure. It also has a thread called Stall_Monitor that is periodic and monitors the condition of the AoA and Climb_Rate sensors and raise a stall warning if certain conditions are met. The thread Operator simulates the pilot. It is a periodic thread that sends a command (Gear_Cmd) at every dispatch to raise or lower the landing gear of the aircraft. The thread Landing_Gear simulates the landing gear subsystem. It receives a command to start a landing gear operation, and is a sporadic thread with a minimum inter-arrival time of 3 seconds. The thread HCI is a human computer interface. It receives a Stall_Warning as an event data of type Integer; Engine_Failure as an event; a landing gear command from the pilot. It may send a landing gear operation request (Gear_Req) to the landing gear subsystem, and receives a landing gear operation acknowledgement (Gear_Ack) from the landing gear subsystem. It is a sporadic thread with a minimum inter-arrival time of 10ms. The graphical representation of Flight Computer system model is given in Figure 11.

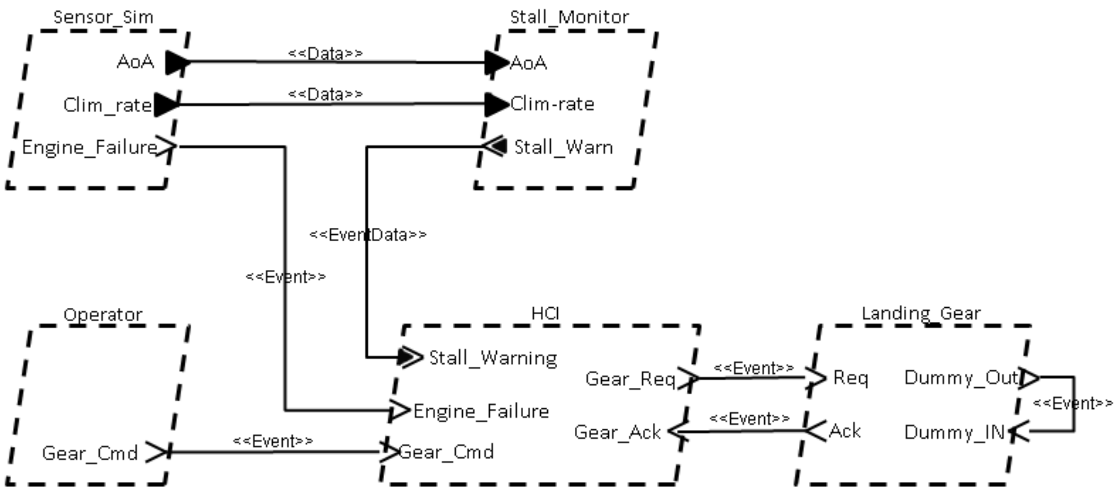


Fig. 11. Flight Computer Architecture

6.1 BIP model

The AADL model of the Flight Computer is transformed into BIP automatically by using our AADL to BIP translation tool. Figure 12 shows the obtained BIP model. This figure represents the BIP atomic components (AADL Threads) and connectors between them. Notice that we omit here the connectors between threads, process and scheduler that are shown in the Figure 9.

The component Dummy_In_Out models the communication between the Dummy_Out and Dummy_In events ports. In the AADL model (Figure 11), these two events are used to control thread reactivation: execution of the Landing_Gear thread is activated by the Dummy_In event; it emits a Dummy_Out event upon completion. Thus, synchronizing these two events ensures periodic activation of this thread.

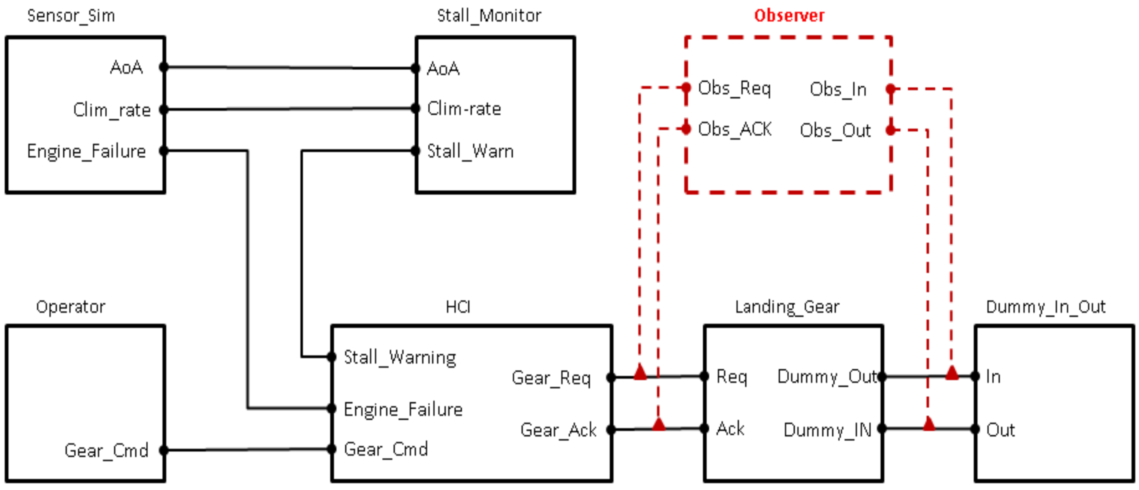


Fig. 12. BIP model for the Flight computer (including observer property, dashed)

6.2 Verification

The model construction methodology applied to this example, opens the way for enhanced analysis and early error detection by using verifications techniques.

Once the model has been generated, two model checking techniques for verification have been applied:

Model checking by Aldebaran: The first technique of verification is deadlock detection by using the tool Aldebaran [12]. Exhaustive exploration by the BIP exploration engine, generates a Labeled Transition System (LTS) which can be analyzed by model checking. e.g, Aldebaran takes as input the LTS generated from BIP and checks for deadlock-freedom. We have checked that the model is deadlock-free.

Model checking with observers: The second technique of verification is by using BIP observers to express and check requirements. Observers allow us to express in a much simple manner most safety requirements. We apply this technique to verify two properties:

- Verification of thread deadlines by using an observer component keeping track of the execution time of threads. If the execution time of a thread exceeds its deadline the observer moves to an error state.
- Verification of synchronization between components: Landing_Gear is sporadically activated bye HCI trough the Req port. When it is activated, it send back an acknowledgement through the ACK port, and possibly reactivates itself through the Dummy_In_Out component. This property can be verified by an observer which monitors the interactions between HCI, landing_Gear and Dummy_In_Out components (Figure 11).

7 Conclusion

The Architecture Analysis and Design Language (AADL) suffers from the absence of concrete operational semantics. In this paper, we address this problem by providing a translation from AADL to BIP, which has an operational semantics formally defined in terms of labelled transition systems. This translation allows simulation of AADL models, as well as application verification techniques, such as state exploration (using IF toolset [13]) or component-based deadlock detection (using Aldebaran [12], and D-Finder tool [11]). The proposed method has been implemented in translation tool, which has been tested on the Flight Computer case study, also presented in this paper. Future work includes incorporating features that will appear in V2.0 of the AADL standard.

References

1. Annex behavior specification sae as5506.
2. http://aadl.enst.fr/arc/doc/.
3. http://gforge.enseeiht.fr/docman/?group_id=37.
4. http://www-verimag.imag.fr/ async/bipmetamodel.php.
5. Sae. architecture analysis & design language (standard sae as5506), september 2004, available at http://www.sae.org.
6. Sei. open source aadl tool environment. http://la.sei.cmu.edu/aadlinfosite/ opensourceaadltoolenvironment.html.
7. Topcased. http://www.topcased.org/.
8. A. Basu, S. Bensalem, M. Gallien, F. Ingrand, C. Lesire, T.H. Nguyen, and J. Sifakis. Incremental component-based construction and verification of a robotic system. In *Proceedings of ECAI'08, Patras, Greece*, 2008.
9. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of SEFM '06, Pune, India*, pages 3–12. IEEE Computer Society, 2006.
10. A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using bip for modeling and verification of networked systems – a case study on tinyos-based networks. In *Proceedings of NCA'07, Cambridge, MA USA*, pages 257–260, 2007.
11. S. Bensalem, M. Bozga, J. Sifakis, and T.H. Nguyen. Compositional verification for component-based systems and application. In *Proceedings of ATVA'08, Seoul, South Korea*, 2008.
12. M. Bozga, J-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the aldebaran toolset. *STTT*, 1:166–183, 1997.
13. M. Bozga, S. Graf, Il. Ober, Iul. Ober, and J. Sifakis. The if toolset. In *Proceedings of SFM'04, Bertinoro, Italy*, volume 3185 of *LNCS*, pages 237–267, September 2004.
14. J. Sifakis G. Gossler. Composition for component-based modeling. *Science of Computer Programming*, 55:161–183, March 2005.
15. M. Poulhiès, J. Pulou, C. Rippert, and J. Sifakis. A methodology and supporting tools for the development of component-based embedded systems. In *13th Monterey Workshop, Paris, France*, volume 4888 of *LNCS*, pages 75–96, 2006.

Deriving component designs from global requirements

Gregor v. Bochmann

School of Information Technology and Engineering (SITE)
University of Ottawa, Canada
bochmann@site.uottawa.ca

Abstract. This paper is concerned with the early development phases of distributed applications, service compositions and workflow systems. It deals with the transformation of a global requirements model, which makes abstraction from the physical distribution of the different system functions, into a system design that identifies a certain number of distributed components. The temporal constraints of the global requirements on the execution of the different activities imply certain coordination messages between the different system components. The paper presents a transformation algorithm that derives, from a given global behavior, the local behaviors for each of the system components including the exchange of coordination messages for the global synchronization of the activities. In contrast to earlier work, strong and weak sequencing is distinguished and the primitive sub-activities included in the global behavior descriptions may be collaborations involving several components.

Keywords. Distributed applications, workflow, model transformations, Activity Diagrams, component design, protocol derivation, distributed system design, Web Services, design derivation.

1 Introduction

Various kinds of system models can be used during the system development process. In this paper, we are concerned with the transformation from a global requirements model, which describes the functional behavior of a distributed system in an abstract manner, to a distributed system design where the different system components are identified and their behavior must be determined such that their interactions give rise to a behavior satisfying the global requirements model. At the design level, the behavior of the different system components are often modeled using communicating state machines or modeling languages such as SDL or UML State Diagrams. The translation from these models into implementation code can be largely automated.

We consider in this paper distributed applications, for instance systems providing communication services, workflow management systems, e-commerce applications, etc. Various notations have been proposed for defining the global requirement models for such system. We mention in particular UML Activity Diagrams, Use Case Maps (UCM), the Process Definition Language (XPDL) of the Workflow Management

Coalition, the Business Process Execution Language (BPEL), and the Web Services Choreography Description Language (WS-CDL) developed by W3C. These different notations contain many common concepts, but also show important differences. They all have in common that the overall workflow behavior can be decomposed into several sub-activities, and further into sub-sub-activities. Most of these notations assume that the basic (primitive) activities in this behavior decomposition are activities that are allocated to a single system component within the architectural design of the system. However, for many of these applications, the basic building blocks of the behavior are activities that are actually collaborations between several system components, for instance a service operation between a client and a server. Therefore we have proposed to use the UML Collaborations as the basic building blocks for constructing global requirement models [1]. Our approach was to use the sequencing operations of UML Activity Diagrams and use Collaborations as the basic activities; the temporal order among these collaborations is then defined by the flow relations of the Activity Diagrams (an example is discussed in Section 2).

Before the transformation into a design model, it is important to define the architectural design and to identify the different system components that are involved in providing the different functions of the system. For each of the primitive collaboration activities identified in the global requirements model, one has to determine which system component will implement each of the collaboration roles involved. This goes hand in hand with the allocation of system resources and is very important for obtaining the desired system performance characteristics. This question of what is the best architectural design, resource allocation and allocation of collaboration roles to different system components is not further developed in this paper. Instead, we concentrate here on the subsequent question: What should be the dynamic behavior of each of the system components in order to coordinate the activities in such a manner that the sequencing rules of the global requirements model will be satisfied.

We note that the same kind of question has been addressed by many papers during the last 10 years in a context where the global requirements are defined in terms of Message Sequence Charts (MSCs) or UML Sequence Diagrams. In this context, one usually wants to describe the behavior of each system component in the form of a state machine. This approach encountered many difficulties; a review of these issues is included in [1]. In many cases, a given MSC execution scenario may only be realizable by the given set of components if at the same time other so-called *implied scenarios* would also be realized [14]. Furthermore, the distributed nature of the design often gives rise to so-called *race conditions* which means that certain messages may arrive before they are expected, or in a different order than expected [16].

These difficulties are increased by the use of **weak** sequencing operators in the description of the global system behavior. We note that **strong sequencing** between two activities A1 and A2 means that all sub-activities of A1 must be completed before any sub-activity of A2 may start. In contrast, **weak sequencing** between A1 and A2 means that each system component locally applies sequencing to the local sub-activities of A1 and A2, that is, a component may start with sub-activities that belong to A2 as soon as it has completed all its local sub-activities that are part of A1. Strong sequencing implies weak sequencing, but not inversely. We note that weak sequencing was introduced in High-Level MSCs (HMSCs) as the normal sequencing operator between different sequence charts. It is also supported in UML Sequence and Interaction Overview diagrams.

We think that weak sequencing is an important concept for modeling abstract requirements of distributed systems, because it requires less synchronization messages than strong sequencing. Therefore we consider in this paper weak and strong sequencing. We use a number of temporal ordering operators, similar to those found in Activity Diagrams, XPDL and BPEL, to build the global requirements model for a system. In this paper, we show how such an abstract model, together with the allocation of collaboration roles to the system components identified by the architectural design, can be automatically transformed into a set of component behavior models. These component models are correct by construction, that is, they will give rise to a global system behavior that satisfies the global requirements model.

The transformation algorithm presented in this paper is inspired by some of our early work under the title "Deriving protocol specifications from service specifications" in the 1980ies [3, 4, 5, 6], where we concentrated our attention on strong sequencing. The main contribution of this paper is the extension of the previous work to requirement specifications that contains weak sequencing. Some inspiration also came from my collaboration with Humberto Nicolás Castejón and Rolv Bræk on the modeling of distributed applications using the concept of collaborations [1, 2] and the discussion of problems that must be solved during the development of the component behaviors.

The paper is structured as follows. In Section 2, we consider the temporal ordering of activities in a global requirements model, present the ordering operators that we assume in this paper and introduce a simple example. The main body of the paper is Section 3. After a review of past work on the transformation from global requirements to component behaviors, we describe in Section 3.1 the principles of our automatic transformation approach. One important question, not addressed by the earlier work mentioned above, is the following: In the case of choices, it is not evident how a component involved in some specific sub-activity may determine when this sub-activity is completed and the next sub-activity (in weak sequence) may be started ? – This problem is solved by the so-called *choice indication messages*. Then in Section 3.2, we present an algorithm that does this transformation automatically. The application of this algorithm to the example of Section 2 is discussed in Section 4. Finally, Section 5 provides our conclusions.

2 Describing composed collaborations and work flow applications

As mentioned above, various notations have been proposed for describing global requirements for distributed applications, workflows, or communication services. We consider here in particular UML Activity Diagrams (AD). They include the following concepts for defining the order of execution of activities: sequential execution, alternative choice, concurrency, as well as loops and partial-order dependencies. In addition, they support interruptible regions of activities which are useful for modeling exception handling and external priority interrupts. ADs also support the explicit modeling of dataflow relationships between different activities and the specification of the type of data exchanged (using UML Class Diagrams). XPDL and BPEL have similar constructs for describing the control flow of applications. All these notations support hierarchical decomposition where an activity shown at one level of abstraction as a basic, non-divisible activity can be described at a more detailed level to be composed out of a number of smaller units with a specific control structure defining the order of execution of these more basic activities. It is our intention to support these same concepts for describing the control structure using the notation introduced below.

We note that most of these notations assume that the basic (primitive) activities in this behavior decomposition are activities that are allocated to a single system component within the architectural design of the system. This is also the case for Message Sequence Charts (MSCs) or UML Sequence Diagrams, where the primitive actions are the sending or reception of messages by specific system components. In contrast, as mentioned in the Introduction, we assume that the basic activities in the description of the overall behavior may be collaborations involving several components.

One may ask the question whether different notations are required for describing the dynamic behavior of the global requirements model, on the one hand, and the behavior of the different system components, on the other hand. In this paper, we use essentially the same behavior expressions to describe both of these behaviors. However, the distinction between weak and strong sequencing disappears when one deals with the behavior of a single component. The operators used for describing the temporal properties of these behavioral models are listed in Table 1. They are closely related to the sequencing operators of UML Activity Diagrams and High-Level MSCs. We distinguish between strong and weak sequence. Following the spirit of "Structured Programming", we restrict ourselves to flow control constructs that have a single entry point and a single exit point.

We write "<name>(R) = C" to indicate that the behavior of a collaboration, called <name>, which involves the set of roles R, is given by the expression C. The expression is composed out of primitive actions, the invocation of collaborations and certain sequencing operators as shown in Table 1. We refer to the sub-expressions C1, C2, and C3 in the table as sub-collaborations of the collaboration C. We note that our notation does not include the equivalent of the Join and Merge operators used in Activity Diagrams. However, the presence of a Merge node is implied at the end of a choice expression, and a Join node is implied at the end of the concurrency construct.

It is possible to invoke a collaboration that has no explicitly defined behavior; in this case, its behavior may be defined by some other formalism, such as a sequence diagram or an implementation in some programming language.

As an example we consider the telemedicine consultation service described in [1]. A patient is being treated over an extended period of time for an illness that requires frequent tests and consultations with a doctor at the hospital to set the right doses of medicine. Since the patient may stay at home and the hospital is a considerable distance away from the patient's home, the patient has been equipped with the necessary testing equipment at home. The patient will call the hospital on a regular basis to have remote tests done and consult with a doctor. A consultation may proceed as follows: The patient calls the telemedicine reception desk to ask for a consultation session with one of the doctors. The receptionist will register the information needed, and then see if the doctor is available (collaboration <registr> below). If the doctor is available, the patient will be assigned to the doctor and the consultation can start. Otherwise, the patient is put on hold, possibly listening to music, until a doctor is available (collaboration <w> below). If the patient does not want to wait any longer, he/she may hang up (action <h-up> below) and call back later.

Table 1: Operators used in behavior expressions

Construct	**Notation**	**Explanation of the semantics**
primitive activity	$\langle action\rangle^{(r)}$	Execution of a local action with name <action> performed by role r
invocation of sub-col.	$\langle subcol\rangle^{(R)}$	Execution of a collaboration with name <subcol> involving the set R of participating roles
strong sequence	$C_1 ;_s C_2$	C_2 is executed after C_1 in **strong** sequence, that is, all actions of C_1 are completed before C_2 can start
weak sequence	$C_1 ;_w C_2$	C_2 is executed after C_1 in **weak** sequence, that is, only local order is enforced by each participating role
choice	C_1 [] C_2	Either C_1 or C_2 is executed; this may be a local choice (that is, the choice is performed by a single role / component) or competing initiatives from several roles; for a more detailed discussion, see [2])
strong while loop	$C_1 *_s C_2$	C_1 is executed zero, one or more times and then C_2 will be executed; more precisely, the behavior starts with a choice between C_1 and C_2 ; if C_1 is executed, there is **strong** sequencing between the end of C_1 and the choice of executing C_1 again or terminating the loop with C_2 ; we assume that the choice is local (performed by a single role).
weak while loop	$C_1 *_w C_2$	As above, except that **weak** sequencing is used between the end of C_1 and the choice of executing C_1 again or terminating the loop with C_2
concurrency	$C_1 \parallel C_2$	C_1 and C_2 are executed concurrently
interruption	C_1 |> C_2 else C_3	C_1 is executed, but may be interrupted by C_2 which represents a choice with priority; C_2 is enabled as soon as C_1 starts. If C_2 does not occur (or occurs when C_1 is already terminated) then C_3 will occur after C_1 (this is the other choice alternative).

This behavior can be described using the operators defined in Table 1 as follows:
$\langle telemed\rangle = \langle registr\rangle^{\{P, R\}} ;_w (\langle w\rangle^{\{P, R\}} \;|\!> \langle h\text{-}up\rangle^{\{P\}} \text{ else } \langle act\rangle^{\{P, R, D\}})$
where $\langle w\rangle^{\{P, R\}} = \langle wait\rangle^{\{P, R\}} *_w \varepsilon$ and
$\langle act\rangle^{\{P, R, D\}} = \langle assign\rangle^{\{R, D\}} ;_w \langle consult\rangle^{\{P, D\}}$
The roles involved in each activity are indicated by the upper indices (P stands for patient, R for receptionist, and D for doctor). This definition of the <telemed> workflow indicates that the registration of the patient is followed by a waiting period <w> that may be empty (ε); this waiting period may be interrupted when the patient hangs up the telephone. The waiting period, if not interrupted, is followed by the <act> sub-collaboration which consists of the weak sequential execution of the assignment of the patient to the doctor followed by the consultation. We note that the detailed interactions involved in each of these activities (or collaborations) are not

specified, and we do not need to know them for what we discuss in this paper. The behavior can also be represented by the UML Activity Diagrams shown in Figure 1.

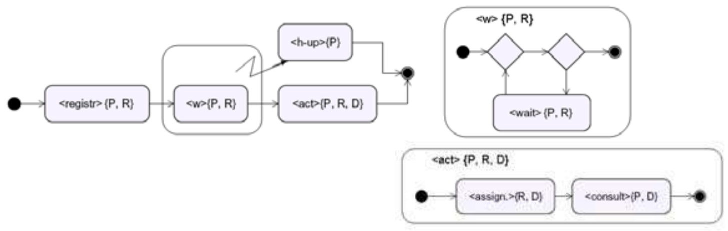


Fig. 1. Dynamic behavior of the Telemedicine collaboration, including two sub-collaborations

3 Deriving component-based designs

Our early work in this area [3, 4, 5] covered behavior expressions containing primitive actions, invocations of behaviors without recursion, strong sequence, choice and concurrency. Coordination messages were introduced for a strong sequence "C1 ;s C2" to ensure that all activities of C1 are completed before any activity of C2 can start. The various messages introduced by the derivation algorithm included a parameter that avoided any ambiguities concerning the choices that were made during the execution of the behavior. A later paper [6] dealt with recursive behavior invocations and interruption.

In the above references, it was assumed that a choice between different branches of execution is always made by a single component. This is called a "local choice". In the case of a "non-local choice" where several components are involved [7], distributed algorithms for making a decision may be introduced, for instance, based on a circulating token. Gouda showed in 1984 [8] how a choice involving competing initiatives from two different components may be resolved by giving priority to one of the parties.

During the last 10 years, much research was concerned with weak sequencing and related race conditions. Most of this work was in the context were the system behavior is defined in terms of MSCs or Sequence Diagrams; and it was pointed out that one sequence diagram, when implemented by a set of components, may necessarily give rise to other so-called “implied sequences” [14]. The difficulties of coordination for distributed behaviors including weak sequencing have been summarized in [2]. An interesting observation was made by Mooij [9, 10] who points out that many race conditions can be avoided by making a distinction between the reception of a message by a component and the consumption of this message by the behavior of a role played by this component. He assumes that received messages are put into a buffer pool from where appropriate messages may be fetched when the destination role is ready to process them. A similar idea is the use of the SDL SAVE construct to reorder the sequence of received messages [15].

In the area of Web Services and workflow management, several approaches have been described for deriving distributed execution environments for services/workflows that are specified in a global (centralized) view. For instance, the decentralized execution of composite Web Services specified in BPEL has been proposed in [18]. Here the global BPEL specification is partitioned into small code fragments which are then combined (based on data flow relations) into several local partitions that are executed on the different servers identified in the original BPEL specification. The resulting implementation is in general more efficient since the number of required messages is reduced. A proposal for workflow fragmentation and distributed execution [19] is also based on data flow and uses a variant of Petri nets for describing the workflow to be performed. The workflow is partitioned into fragments of which the first is executed locally and the others may be distributed to other servers. The choice of these servers can be performed dynamically during the execution of the current fragment. A theoretically oriented paper [20] considers a formalization of WS-CDL for the specification of the global behavior and the π–calculus for the behaviors of the components. We note that these approaches consider that the basic activities to be performed can be allocated to a single component (server). Therefore they cannot deal with the more general situation considered in this paper, where the basic activities are collaborations that may involve, each, several collaborating components.

3.1 Proposed derivation method

The derivation method described here uses the following ideas described previously: (a) coordination messages for strong sequencing [3, 4, 5], (b) the idea that messages should have an identifier that indicates to which sub-expression of the behavior expression they belong (particular methods of obtaining such an identifier were proposed by Nakata [11], and for Application Protocols in the ASN.1 standard), and (c) the idea of buffering received messages until they are processed, as proposed in [9, 10]. The proposed derivation method extends the previous work by providing a method to deal with weak sequencing. It also introduces the treatment of loops and a particular form of interruption. For the treatment of non-local choices, the reader is referred to [2].

The main ideas underlying the proposed derivation method, specifically for dealing with weak sequencing, can be summarized as follows:

1. Each role knows which sub-collaborations are currently active. Message re-ordering at reception is used to accept only those messages that relate to active collaborations.
2. It is assumed that sub-collaborations that may be concurrently active have disjoint sets of messages that can be received by a given role; or simply, that their message sets are disjoint.
3. At a given role, each sub-collaboration is in one of the following phases: (1) **inactive** (messages for this sub-activity are not accepted), (2) **enabled** (the role is not a starting role, the messages of this sub-activity are accepted), (3) **active** (local activities for this sub-activity have started, messages are accepted). We say that a sub-collaboration **ends** when the role knows that no

further actions pertaining to this sub-collaboration are required. When a sub-collaboration ends, it goes back into the **inactive** phase.

4. The transition from *inactive* to *enabled* occurs when the "previous" sub-collaboration ends. If the role is a starting role, it may immediately go into the *active* state. A non-starting role enters the *active* state when it receives (and accepts) the first message pertaining to this sub-collaboration. When the sub-collaboration *ends*, the "following" sub-activity goes into the *enabled* or *active* state.
5. It is therefore important that each role knows when an active collaboration ends. This happens when the final action (for this role) is performed. If the role is participating, it should know when all actions pertaining to this sub-collaboration have been performed (**termination decision**). If it is not participating, there is no point in doing anything.
6. If the sub-collaboration contains no choice, then each role knows what actions must be locally performed. The *termination decision* is easy: the sub-collaboration ends when all these actions have been performed. If the sub-collaboration consists of a choice C1 [] C2, there are the following cases:
    - The role participates in both alternatives: choice propagation is assured by the disjointness of the message sets of the two alternatives. The participating role will know which alternative is performed and will therefore know which actions must be performed.
    - The role does not participate in any alternative: there is no participation at all.
    - The role participates in C1 but not in C2 (or inversely): If C1 is chosen, there is no problem. If C2 is chosen, we have to introduce a special kind of coordination message sent to this role by a role participating in C2 which indicates that C2 was chosen. We call this message a *choice indication message*. On the reception of this message, the given role can consider that the choice has ended.

The above discussion indicates that we have to identify for each collaboration or sub-collaboration the following items which are defined based on the partial order between the actions that compose the collaboration:

- **Special kinds of actions of a collaboration**
    - *Initial action(s):* An action of a collaboration is initial if there is no other action in that collaboration that precedes it.
    - *Final action(s):* An action of a collaboration is final if there is no other action in that collaboration that succeeds it.
    - *Last action(s)* of a given role: During the execution of a collaboration, an action is a last action for a given role if the action is performed by that role and there is no other action in that collaboration that must be performed by that role after the given action.
- **Different roles involved in a collaboration**
    - *Starting* role: this is a role that performs an initial action of the collaboration or an initial action of an initial sub-collaboration.
    - *Terminating* role: this is a role that performs a final action of the collaboration or a final action of a final sub-collaboration.

- *Participating* role: this is a role that executes a primitive action of the collaboration or of a sub-collaboration. This includes the starting and terminating roles.

Table 2 shows how the sets of starting, terminating and participating roles are calculated for a behavior expression depending on the sequencing operators used.

Table 2: Rules for calculating the starting, terminating and participating roles

Operator	**Starting roles (SR)**	**Terminating roles (TR)**	**Participating roles (PR)**
<action>$^{(r)}$	{r}	{r}	{r}
<subcol>$^{(R)}$	SR(<name>)	TR(<name>)	PR(<name>) = R
$C_1 ;_s C_2$	$SR(C_1)$	$TR(C_2)$	$PR(C_1)$ U $PR(C_2)$
$C_1 ;_w C_2$	$SR(C_1)$ U ($SR(C_2)$ - $PR(C_1)$)	$TR(C_2)$ U ($TR(C_1)$ - $PR(C_2)$)	$PR(C_1)$ U $PR(C_2)$
C_1 [] C_2	$SR(C_1)$ U $SR(C_2)$	$TR(C_1)$ U $TR(C_2)$	$PR(C_1)$ U $PR(C_2)$
$C_1 *_s C_2$	$SR(C_1) = SR(C_2)=$ {r}	$TR(C_2)$; $SR(C_1)$ if $C_2= \varepsilon$	$PR(C_1)$ U $PR(C_2)$
$C_1 *_w C_2$	*as above*	$TR(C_2)$ U ($TR(C_1)$ - $PR(C_2)$)	$PR(C_1)$ U $PR(C_2)$
$C_1 \Vert C_2$	$SR(C_1)$ U $SR(C_2)$	$TR(C_1)$ U $TR(C_2)$	$PR(C_1)$ U $PR(C_2)$
$C_1 \vert\!> C_2$ else C_3	$SR(C_1)$	$TR(C_2)$ U $TR(C_3)$	$PR(C_1)$ U $PR(C_2)$ U $PR(C_3)$

Before we can derive the behavior of the distributed system components that should implement the actions defined by the collaboration behavior, we have to determine how the different roles defined in the behavior of the collaboration are allocated to the different system components. In general, each system component should have some role to play, but several behavior roles may be allocated to the same system component. We assume in the following that a function Alloc() defines for each role the system component to which it is allocated.

After having calculated the starting (SR), terminating (TR) and participating (PR) roles for the collaboration and each of its sub-collaborations, we then can derive the behavior for each system component as follows. Basically, the control flow of the behavior of each system component follows the control flow of the collaboration behavior; it is obtained from the global behavior specification of the collaboration "by projection" onto the particular component. This means that actions not local to the component in question are dropped. Therefore any sub-collaboration for which no participating role is allocated to the component in question will also be dropped.

In addition, the following coordination messages between different system components are introduced (for details, see Table 3):

– *Flow message* for coordinating strong sequencing, abbreviated fm(x) or fim(x, i); each message includes a parameter x which indicates to which strong sequencing construct the message belongs within the syntactical structure of the overall collaboration behavior, as originally proposed in [3].
– *Choice indication message* for propagating the choice to a component that does not participate in the selected alternative, abbreviated cim(y) where y indicates to which choice construct the message refers; note that such a message is only required if the destination component is involved in some activities following the choice.

– *Interrupt and interrupt enable messages* for coordinating the interruption of an ongoing activity, abbreviated im(z) and iem(z), respectively, where z indicates to which interrupt construct the message refers.

3.2 Algorithm for deriving component behaviors from global behavior

In the following, we define an algorithm that realizes the derivation method explained in Section 3.1. We assume that the overall workflow is defined by a main-collaboration and several sub-collaborations, identified by their name, which are invoked by the main-collaboration or some activated sub-collaboration. Each of these collaborations is defined by a behavior expressions C which is formed by primitive actions, sub-collaboration invocations and the operators introduced in Table 1. For each of the components c that implements the roles of these collaborations, we define in the following a translation functions T_c that translates the behavior expressions of the collaborations into local behavior expressions to be performed by the component in question. These local behavior expressions will include those primitive actions of the collaborations that are performed by the component in question, in addition to the sending and receiving of coordination messages as required by the behavior expressions. Overall, the syntactic structure of the resulting behavior expressions for all these components resembles the syntactic structure of the original expression of the global collaboration behavior.

In the following we make the assumption that all choices are local. We note that certain standard approaches to solving non-local choices could be easily integrated with our derivation algorithm. However, as explained in [1, 2], the nature of non-local choices may vary a lot in practice and it appears necessary to allow for ad-hoc solutions to fit the specific requirements in particular cases

Table 3 contains the definition of the translation function T_c (C) that defines for a given global behavior expression C the behavior of the system component c. It is defined recursively by the rules in the table. The resulting component behavior expression is constructed using the same sequencing operators as for describing the global behavior, however, since the behavior is performed locally by a given component, there is no point in making a distinction between weak and strong sequencing. We simply use the operator ";" to denote sequential execution.

The text defining the translation function in the table uses a notation similar to Java Server Pages, namely a mixture of text that represents the generated specification of the component behavior, and of text that represents actions and decisions to be performed during the translation. The latter is written in italics. We also include some comments (written between "(*" and "*)") and notes for making the definition of the translation more readable.

As mentioned at the beginning of Section 3.1, the parameters of the coordination messages and the buffering of received messages before their consumption are important elements for the correct operation of the distribution system derived by the algorithm of Table 3. We make the following assumptions:

1. Each coordination message contains the following parameters: (a) source role, (b) destination role, (c) name of sub-collaboration it belongs to, (d) the particular sequencing operator instance it refers to within the global behavior

expression of the given sub-collaboration – these are the parameters named x, y, and z in Table 3. As noted earlier, the parameters (c) and (d) above are important for non-ambiguous choice propagation (see also [3, 4, 5]). In addition, the messages also need addressing information in order to be transmitted through the network to the right computer and the responsible application.

2. The reception of coordination messages proceeds in two steps: When a message is received by a component, it is first placed into a buffer pool, called *receive-buffer*. It will be "consumed" from the *receive-buffer* only when the behavior expression generated for the component according to Table 3 foresees the execution of a receive statement for a message of the specific type and parameter values. The message parameters mentioned above, and the additional parameter mentioned under point 4 below are used to determine whether a message in the *receive-buffer* is "receivable". If no receivable message is in the *receive-buffer*, the execution of the local behavior will wait until such a message arrives.
3. The flow messages used within an interrupt construct, abbreviated fim(x, i), have an additional Boolean parameter i that indicates whether an interrupt was successful.
4. The execution of a weak while loop, say $C_1 *_w C_2$, within the distributed environment may lead to situations where the component deciding the looping conditions, say c_1 , may already have performed several iterations of C_1 while another component c_2 may have only started the first iteration (as shown in Figure 9(c) of [1]). When c_2 receives a flow message indicating the beginning of C_2 , it is important that c_2 can determine whether C_2 should be started or whether more executions of C_1 should first be performed. Therefore we include an additional parameter, say n, in all flow messages that are part of the coordination within C_2 , which contains the number of times that C_1 has been executed. For more details, see [17].

Table 3: Definition of the translation function T_c for component c

Operator	**Definition of T_c**
C = <action>$^{(r)}$	T_c (C) = *if Alloc(r) = c then* <action> *else* ε Note: ε is the empty string and means that no actions need to be performed.
C = invoke <subcol>$^{(R)}$	T_c (C) = *if c in Alloc(R) then* invoke <subcol> *else* ε
C = $C_1 ;_s C_2$	T_c (C) = T_c (C_1) ";" SFM(C_1 , C_2) ";" RFM(C_1 , C_2) ";" T_c (C_2) *where* SFM(C_1 , C_2) = *if c in Alloc(TR(C_1)) then* "send fm(x) to all c' in (Alloc(SR(C_2)) – {c})" *and* RFM(C_1 , C_2) = *if c in Alloc(SR(C_2)) then* "receive fm(x) from all c' in (Alloc(TR(C_1)) – {c} " Note: The term "– {c}" avoids that flow messages are sent to the component itself.
C = $C_1 ;_w C_2$	T_c (C) = T_c (C_1) ";" T_c (C_2)

$C = C_1$ [] C_2	$T_c(C) = DOcim_c(C_1, C_2)$ [] $DOcim_c(C_2, C_1)$ where $DOcim_c(C_1, C_2) =$ if *c in Alloc(PR(C_1)) then* "(T_c (C_1)" *if c is responsible for cim then* "|| send cim(y) to all c' in (Alloc(PR(C_2)) - Alloc(PR(C_1))))" ; else *if c in (Alloc(PR(C_2) - Alloc(PR(C_1))) then* "receive cim(y)" Note: The function $DOcim_c(C_1, C_2)$ generates code for performing C_1, and looks after the transfer of choice indication messages from some component participating in C_1 to those components not participating in C_1, but in C_2.
$C = C_1 *_s C_2$	We assume Alloc(SR(C_1)) = {r}, and Alloc(SR(C_2)) = {r} or $C_2 = \varepsilon$. $T_c(C)$ ="(" $T_c(C_1)$ ";" SFM(C_1 , C_1) ";" RFM(C_1 , C_1) ")* ; (" $T_c(C_2)$ *if c=r then* "|| send cim(y) to all c' in PR" *if c in PR then* "|| receive cim(y) from r" ")" *where* PR = Alloc(PR(C_1)) - Alloc(PR(C_2)) – {r}
$C = C_1 *_w C_2$	As above, except that the SFM and RFM constructs are absent
$C = C_1 \parallel C_2$	$T_c(C) = T_c(C_1) \parallel T_c(C_2)$
$C = C_1$ |> C_2 else C_3	We assume that C_2 has the form " <action>$^{(r)}$ $;_s$ C_2' ". T_c (C) = NormalBeh ||* InterruptBeh . (see note below) These two parts communicate within each component using the following boolean local variables which are initially false: Interr : an interrupt occured (but it may have occurred too late) Interrupted : the normal behavior has been interrupted In addition, a local variable I-Enabled is used by the InterruptBeh part. The action "wait(x)"waits until the expression x becomes true. **NormalBeh** = *if c in Alloc(PR(C_1)) then* "(T_c (C_1) |> (wait(Interr); Interrupted := true;) else ε);" *if c in Alloc(TR(C_1)) then* "send fim(x, Interrupted) to all c' in SR" *if c in SR then* "(for all c' in (Alloc(TR(C_1))–{c}) do (receive fim(x, i) from c'; if i then Interrupted := true;); if not Interrupted then $DOcim_c$ (C_3, C'$_2$);) ||* (wait(Interrupted); $DOcim_c$ (C'$_2$, C_3))) " *else* " ($DOcim_c$ (C'$_2$, C_3) [] $DOcim_c$ (C_3, C'$_2$)); " *where* SR = (Alloc(SR(C'$_2$)) U Alloc(SR(C_3))) –{c} **InterruptBeh** = *if c = r then (* *if c in (Alloc(SR(C_1)) then* "I-Enabled := true; " *else* "for all c' in (Alloc(SR(C_1))–{c}) do (receive iem(z); I-Enabled := true) || (wait(I-Enabled); <action> (* this may never happen *) ; Interr := true; send im(z) to all c' in (Alloc(PR(C_1)) - r) ;) " *else (* c not equal r *) (* *if c in Alloc(SR(C_1)) then "*send iem(z) to r; " *if c in Alloc(PR(C_1)) then* "(receive im(z) from r (*may not happen *); Interr := true;)"

The expression "C1 ||* C2" has the meaning that the two sub-expressions C1 and C2 are executed in parallel, but the whole construct terminates as soon as C1 terminates.

4 Application to the Telemedicine example

Referring to the example discussed in Section 2, let us assume that the roles P (patient), R (receptionist) and D (doctor) are to be implemented on three different components, also called P, R and D, respectively. In the following we explain how the algorithm described in Section 3 can be used to derive the behavior of these components such that they realize the correct coordination of activities among these three components.

We assume that the starting and terminating roles of the basic activities are as follows: SR(<registration>) = TR(<registration>) = {P}; SR(<wait>) = {R}; TR(<wait>) = {P}; SR(<h-up>) = TR(<h-up>) = {P}; SR(<assign>) = TR(<assign>) = {R}; SR(<consult>) = {D}; TR(<consult>) = {P, D}. Using Table 2, this leads to the starting and terminating roles of the sub-activities <w> and <act> as follows: SR(<w>) = TR(<w>) = {R}; SR(<act>) = {R}; TR(<act>) = {P, D, R};

Let us first determine the behavior for the sub-activities <w> and <act> at each of the three components:

T_P (<w>) = T_P (<wait>) * ; receive cim(y) from R
T_P (<act>) = T_P (<consult>) (* P is not involved in <assign> *)
T_R (<w>) = T_R (<wait>) * ; send cim(y) to P
T_R (<act>) = T_R (<assign>) (* R is not involved in <consult> *)
T_D (<w>) = ε
T_D (<act>) = T_D (<assign>) ; T_D (<consult>)

Now let us determine the behaviors of the three components for the <telemed> activity. Applying the rules of Table 3, we obtain the following behaviors for all components c = P, R or D:

T_c (<telemed>) = T_c (<registr>) ; T_c (<w> |> <h-up>; ε else <act>)
= T_c (<registr>) ; (NormalBeh $_c$ || InterruptBeh $_c$)

where T_D (<registr>) = ε and the behaviors NormalBeh $_c$ and InterruptBeh $_c$ are defined as follows:

NormalBeh $_P$ = (T_P (<w>) |> (wait(Interr); Interrupted := true;) else ε);
(receive cim(y) from R [] T_P (<act>))
InterruptBeh $_P$ = receive iem(z) from R; <h-up>; Interr := true; send im(z) to R
NormalBeh $_R$ = (T_R (<w>) |> (wait(Interr); Interrupted := true;) else ε);
(receive fim(x, i) from P; if i then Interrupted := true; if not Interrupted
then T_R (<act>)) || (wait(Interrupted); send cim(y) to D and P)
InterruptBeh $_R$ = send iem(z) to P; receive im(z) from P; Interr := true
NormalBeh $_D$ = T_D (<act>) [] receive cim(y) from R
InterruptBeh $_D$ = ε

By substituting the behaviors of the sub-activities <w> and <act> given above, we obtain three behavior expressions for the three system components P, R and D. These expressions include the local behaviors of the primitive collaborations <wait>, <assign> and <consult> and are independent of their particular nature; the expressions only depend of the sets of starting, terminating and participating roles given above. We note that these behaviors can also be represented by UML Activity Diagrams. Further details are given in [17]. If the behaviors of the primitive collaborations are also given, for instance in the form of simple Sequence Diagrams, a complete description of each component behavior can be obtained by substitution. An example is discussed in [17] in more detail, including possible execution scenarios.

6 Conclusions

We assume that the system design for a distributed system consisting of several separate components can be developed in the following steps:

1. Construction of a requirements model including the specification of the global behavior of the system in terms of basic activities and their temporal ordering.
2. Through architectural and non-functional requirements, a certain number of separate system components are identified; each of the activities identified at the requirements level is either allocated to one of these components, or performed as a collaboration among several components.
3. Based on the global behavior of the requirements, the identified components and a more detailed description of the basic activities, the distributed system design is developed which defines the behavior of each of the system components including the messages required for realizing the collaborations and for ensuring the global coordination of all activities among the different system components.

We have shown in this paper how the third step can be automated, assuming that the global behavior is given in a suitable modeling language. The modeling language supported by our design derivation algorithm described in Section 3 supports most of the concepts found in UML Activity Diagrams. This includes stepwise refinement where the behavior of a given activity is further detailed in terms of sub-activities and their ordering constraints, described as a separate activity diagram. In addition, a distinction between weak and strong sequencing can be made in the requirements model. We plan to prove the correctness of the algorithm, as discussed in [17].

We believe that this approach to the automatic derivation of distributed system designs is useful in many fields of application, including distributed workflow management systems, service composition for communication services, e-commerce applications, or Web Services.

We plan to work on the implementation of the here proposed derivation algorithm in a tool environment and on the extension of the algorithm to support more general order relationships including data flow.

Acknowledgements: I would like to thank Rolv Braek and Humberto Nicolás Castejón for many interesting discussions on the problems and issues related to this paper, and Fedwa Laamarti and Toqeer Israr for suggesting improvements to an earlier version of this paper.

References

[1] H. Castejón , G.v. Bochmann, R. Bræk, Using Collaborations in the Development of Distributed Services, submitted for publication.

[2] H. Castejón, R. Bræk, G.v. Bochmann, Realizability of Collaboration-based Service Specifications, Proceedings of the 14th Asia-Pacific Soft. Eng. Conf. (APSEC'07), IEEE Computer Society Press, pp. 73-80, 2007.

[3] G. v. Bochmann and R. Gotzhein, Deriving protocol specifications from service specifications, Proc. ACM SIGCOMM Symposium, 1986, pp. 148-156.

[4] R. Gotzhein and G. v. Bochmann, Deriving protocol specifications from service specifications including parameters, ACM Transactions on Computer Systems, Vol.8, No.4, 1990, pp.255-283.

[5] F. Khendek, G. v. Bochmann and C. Kant, New results on deriving protocol specifications from services specifications, Proc. SIGCOMM'89, July 1989, in Computer Communications Review Vol.19 no.4, pp. 136-145.

[6] C. Kant, T. Higashino and G. v. Bochmann, Deriving protocol specifications from service specifications written in LOTOS, Distributed Computing, Vol. 10, No. 1, 1996, pp.29-47.

[7] H. Ben-Abdallah and S. Leue, "Syntactic detection of process divergence and non-local choice in Message Sequence Charts", *Proc. 2nd* Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), 1997

[8] M. G. Gouda and Y.-T. Yu, Synthesis of communicating Finite State Machines with guaranteed progress, IEEE Trans on Communications, vol. Com-32, No. 7, July 1984, pp. 779-788.

[9] Mooij, Arjan J., Goga, Nicolae, & Romijn, Judi. 2005. Non-local Choice and Beyond: Intricacies of MSC Choice Nodes. *Pages 273–288 of: FASE.*

[10] Mooij, Arjan, Romijn, Judi, & Wesselink, Wieger. 2006. Realizability criteria for compositional MSC. *In: Proc. of 11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06)*. LNCS, vol. 4019. Springer.

[11] A. Nakata, T. Higashino and K. Taniguchi, Protocol synthesis from context-free processes using event structures, in Proc. of 5th Int'l Conf. on Real-Time Computing Systems and Applications (RTCSA'98), Hiroshima, Japan, IEEE Computer Society Press, pp.173-180, Oct. 1998.

[12] H. Kahlouche and J. J. Girardot, "A Stepwise Requirement Based Approach for Synthesizing Protocol Specifications in an Interpreted Petri Net Model," Proc. INFOCOM'96, pp. 1165–1173, 1996.

[13] H. Yamaguchi, K. El-Fakih, G. v. Bochmann and T. Higashino, Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets, Distributed Computing, Vol. 16, 1 (March 2003), pp. 21-36.

[14] Alur, Rajeev, Etessami, Kousha, & Yannakakis, Mihalis. 2000. Inference of message sequence charts. *Pages 304–313 of: 22nd International Conference on Software Engineering (ICSE'00)*.

[15] F. Khendek and X. J. Zhang, "From MSC to SDL: Overview and an application to the autonomous shuttle transport system", *Proc. 2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, LNCS, vol. 3466, 2005.

[16] R. Alur, G. J. Holzmann and D. Peled, "An analyzer for Message Sequence Charts", Software - Concepts and Tools, 17(2), 70–77, 1996.

[17] G.v. Bochmann, "Deriving component designs from global service and workflow specifications", submitted for publication.

[18] M.G. Nanda, S. Chandra and V. Sankar, "Decentralizing execution of composite Web Services", Proc. OOPSLA'04 (ACM), Vancouver, Canada, 2004.

[19] W. Tan and Y. Fan, "Dynamic workflow model fragmentation for distributed execution", Computers in Industry, Vol. 58, pp. 381-391 (2007).

[20] M. Carbone, K. Honda and N. Yoshida, "Structured communication-centred programming for Web Services", ESOP'2007.

Scalable Models Using Model Transformation [⋆]

Thomas Huining Feng and Edward A. Lee

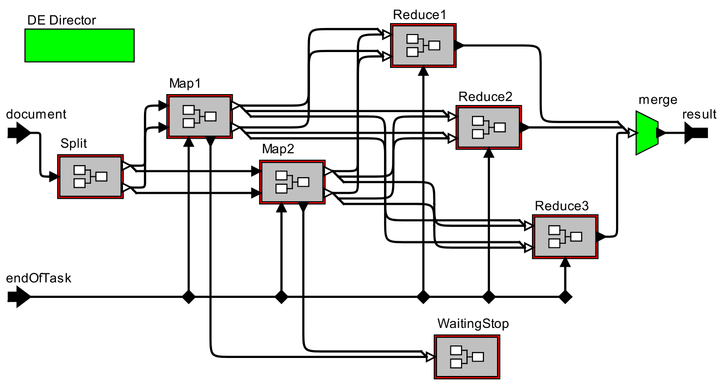
Center for Hybrid and Embedded Software Systems
EECS, University of California, Berkeley
Berkeley, CA 94720, USA
{tfeng,eal}@eecs.berkeley.edu

Abstract. Higher-order model composition can be employed as a mechanism for scalable model construction. By creating a description that manipulates model fragments as first-class objects, designers' work of model creation and maintenance can be greatly simplified. In this paper, we present our approach to higher-order model composition based on model transformation. We define basic transformation rules to operate on the graph structures of actor models. The composition of basic transformation rules with heterogeneous models of computation form complex transformation systems, which we use to construct large models. We argue that our approach is more visual than the traditional approaches using textual model descriptions. It also has the advantage of allowing to dynamically modify models and to execute them on the fly. Our arguments are supported by a concrete example of constructing a distributed model of arbitrary size.

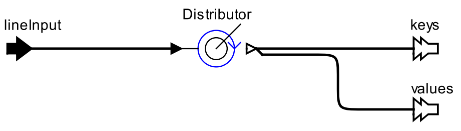
1 Introduction

We take an actor-oriented approach to the design of embedded systems. A model consists of actors as the basic building blocks, which implement functions that map signals at their input ports to signals at their output ports. The wiring between output ports and input ports represents transmission of unaltered signals. In a hierarchical design, models may contain models, in which case the contained models themselves act as actors. The execution semantics are defined by the director of the model, if it has one, or otherwise the director of the containing model. Each director implements a model of computation (MoC). Examples of MoCs include DE (Discrete Event), SDF (Synchronous Dataflow), FSM (Finite State Machine), and PN (Karn Process Network). A heterogeneous model uses various MoCs at different levels of its hierarchy. This proves extremely flexible for system design, because model designers can freely choose a convenient MoC for any part of the model [1].

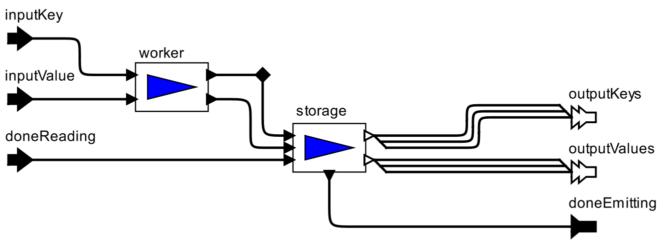
[⋆] This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab



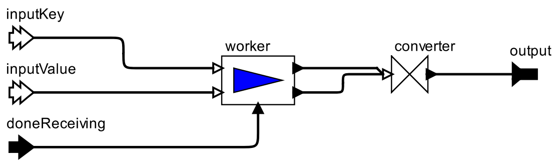
(a) Top level of the MapReduce model



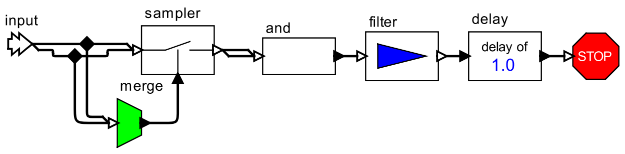
(b) Internal design of the Split actor



(c) Internal design of the Map actor



(d) Internal design of the Reduce actor



(e) Internal design of the WaitingStop actor

Fig. 1. MapReduce model with 2 Map machines and 3 Reduce machines

In recent practice we have seen large-scale models containing thousands or even millions of actors. One concrete example is the model for distributed word-counting created using the MapReduce programming paradigm as described by Dean and Ghemawat in Google [2]. The model is designed to utilize hundreds of computers available in a large cluster to count the occurrences of each word in a huge number of web documents. We have created a simplified demo using 5 worker machines in the Ptolemy II modeling and simulation environment [3], as shown in Fig. 1. Two of the machines (modeled by actors) provide the Map functionality and three provide Reduce. The Split actor, located either on a central computer or on any of the worker machines, distributes the key-document pairs received at its input port to the Map machines. The Map machines map words in the input documents onto word-number pairs. In this case, the numbers in those pairs are always 1, each denoting a single occurrence of a word. Those pairs are then sent to the Reduce machines designated by the hash code of the words. Therefore, pairs containing the same word are always delivered to the same Reduce machine. The Reduce machines then count the pairs for each word that they receive, and send the result to the Merge actor for output. When all input documents are processed, the WaitingStop actor receives a true-valued input, and terminates the execution.

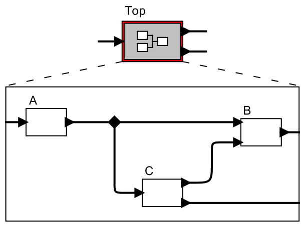
It is hard to imagine manually constructing one such model for a large number of worker machines. The complexity of the work grows sharply as the number of actors increases. Even if the model can be constructed manually, it is still extremely difficult to modify or maintain the design. We are thus motivated to explore an approach to automated model construction and modification.

In our prior work, we have developed Ptalon as a declarative language for higher-order model composition [4]. Textual model descriptions can be written by designers to manipulate model fragments as first-class objects, and to compose them to generate large models. As an example, the same MapReduce model is constructed with Ptalon [5]. It is shown that the size of the Ptalon description does not grow with the increase of worker machines used by the model. This is because the number of worker machines is defined as an integer parameter that can be set by the user.

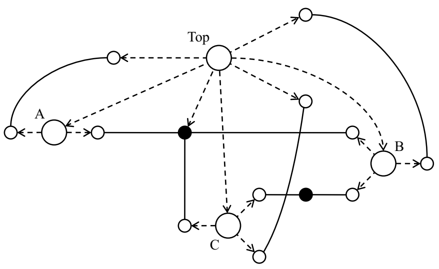
In our recent effort on higher-order model composition, we aim to provide a more straightforward and user-friendly mechanism for constructing models. We view models as attributed graphs, and employ graph transformation technique. We have implemented a tool that supports a visual language for specifying transformations. The visual language is very close to the language that model designers use to manually create models. This removes the need for requiring model designers to learn new languages.

We envision a variety of applications for our technique. In this paper, we present an example for automated model construction and execution. Other applications include model optimization, refactoring, structural parametrization, and workflow automation.

(AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.



(a) An example of a hierarchical model



(b) Attributed graph representation

Fig. 2. Hierarchical model and its representation in attributed graph

This technique differs from Ptalon in the following ways.

1. Models are constructed with graph transformation rules that have a visual representation, whereas in Ptalon, textual descriptions are used.
2. Transformations can be applied to existing models as incremental modifications statically or dynamically.
3. A hierarchical heterogeneous model can be used to control the application of multiple transformations.

The following sections are organized as follows. In Section 2, we present models as graphs and define basic transformations. In Section 3, we discuss using a model to control basic transformations to form a complex transformation. We construct a MapReduce model with transformation as an example in Section 4. We study the related work in Section 5, and conclude our discussion in Section 6.

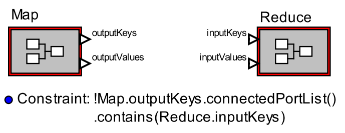
2 Model Transformation Based on Graph Transformation

Fig. 2 shows how a hierarchical model can be represented with an attributed graph. In Fig. 2(b), vertices represent actors, ports, and relations in the model. The styles of the vertices denote their types encoded with attributes, as will be discussed later. Actors are represented by big circles, ports represented by small hollow circles, and relations by filled dots. There are two types of edges. Dashed lines represent containment relationship, where the end vertices are semantically contained by the start vertices. Solid lines represent connections between ports. To represent a bidirectional connection between ports, we use two reversed directed edges.

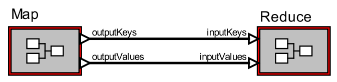
This alternative representation of models allows us to directly apply graph transformation techniques [6] to modify model structures.

2.1 Visual Representation of Transformation Rules

We use a visual syntax to specify transformation. This syntax is inspired by triple graph grammar [7]. A transformation is defined by a *transformation rule*,


Map
Reduce
outputKeys
outputValues
inputKeys
inputValues
● Constraint: !Map.outputKeys.connectedPortList()
.contains(Reduce.inputKeys)

(a) Pattern



(b) Replacement

	Pattern	Replacement
1	Map	Map
2	Reduce	Reduce

(c) Correspondence

Fig. 3. A transformation rule for connecting a Map and a Reduce

which is similar to a rewrite rule in a context-free grammar. It can be used to match a subgraph in a given graph, and to replace that subgraph with a replacement graph. We specify a transformation rule with three components: a *pattern*, a *replacement*, and the *correspondence* between objects in those two.

Fig. 3 shows a transformation rule designed for our MapReduce example, which we will further discussed in Section 4. This rule creates connections between the output ports of a Map actor and the input ports of a Reduce actor. Repeatedly applying the rule results in having all the Map actors and Reduce actors connected in this way. In the pattern, two *matchers* aim to match two distinct actors in the given model. The names of the matchers are insignificant and need not be the same as those of the matched actors. Without considering the constraint, the two matchers in the pattern match any two such actors: one with output ports named “outputKeys” and “outputValues,” and the other with input ports “inputKeys” and “inputValues.” The constraint requires that the ports of the two actors not be connected before the transformation is applied. This avoids creating excessive connections between the same pairs of ports.

In the replacement, the two matchers are preserved, meaning that the matched actors should be kept after transformation. Two connections are to be created between their ports, because those connections (along with the hidden relations on them) do not exist in the pattern. In general, designers of transformation rules can specify adding or deleting objects by editing the replacement as they wish. Note that the names of the matchers in the replacement need not be the same as those in the pattern, because the third component, the correspondence table, establishes the relations between the two graphs. In this case, the correspondence table states that the “Map” object in the pattern corresponds to the “Map” object in the replacement, and the “Reduce” object in the pattern corresponds to the “Reduce” object in the replacement. For brevity, we do not show correspondence relations between other types of vertices such as ports and relations.

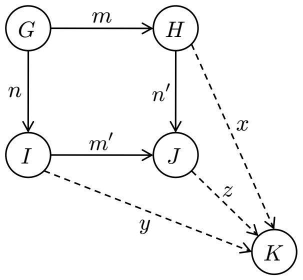
2.2 Formal Definition of Graph Transformation

Our graph transformation is defined as a modified version of the double-pushout approach introduced in [8] and reviewed in [9]. For completeness, we first review that approach here.

A graph G is a tuple $\langle V_G, E_G \rangle$, where V_G is the set of vertices in G and $E_G \subseteq V_G \times V_G$ is the set of edges. Graph G is a *subgraph* of graph H if $V_G \subseteq V_H$ and $E_G \subseteq E_H$.

A *graph morphism*, or simply *morphism*, from graph G to H is a total function $m : V_G \to V_H$, such that for any $v_1, v_2 \in V_G$, if $(v_1, v_2) \in E_G$, then $\big(m(v_1), m(v_2)\big) \in E_H$. We denote this morphism with $G \xrightarrow{m} H$. For any vertex $v \in V_G$, we say v *matches* v' in m if $m(v) = v'$. For any edge $(v_1, v_2) \in E_G$, we say (v_1, v_2) *matches* (v'_1, v'_2) in m if $m(v_1) = v'_1$ and $m(v_2) = v'_2$. If m is an injective function, then we say G is *isomorphic to a subgraph of* H, or G *matches a subgraph of* H.

The *composition* of $G \xrightarrow{m} H$ with $H \xrightarrow{n} I$ is $G \xrightarrow{n \circ m} I$, where $n \circ m : V_G \to V_I$ is the composition of function $m : V_G \to V_H$ and $n : V_H \to V_I$.



$$\begin{tikzcd}
G \arrow[r, "m"] \arrow[d, "n"'] & H \arrow[d, "n'"] \arrow[ddr, dashed, "x"] & \\
I \arrow[r, "m'"] \arrow[drr, dashed, "y"'] & J \arrow[dr, dashed, "z"] & \\
 & & K
\end{tikzcd}$$

Fig. 4. Pushout of graph morphisms

Given graphs G, H, I, and morphisms $G \xrightarrow{m} H$ and $G \xrightarrow{n} I$ as depicted in Fig. 4, a *pushout* is a tuple $\langle J, I \xrightarrow{m'} J, H \xrightarrow{n'} J \rangle$, in which J is a graph and morphisms $I \xrightarrow{m'} J$ and $H \xrightarrow{n'} J$ satisfy the following conditions:

1. $n' \circ m = m' \circ n$, and
2. For any graph K with morphisms $H \xrightarrow{x} K$ and $I \xrightarrow{y} K$ satisfying $x \circ m = y \circ n$, there exists a unique $J \xrightarrow{z} K$ satisfying $z \circ n' = x$ and $z \circ m' = y$.

A *transformation rule* T, denoted by $\langle P \xleftarrow{m} K \xrightarrow{n} R \rangle$, consists of graphs P, K and R, and injective morphisms $K \xrightarrow{m} P$ and $K \xrightarrow{n} R$. P is called the *pattern graph* (or left-hand side). R is the *replacement graph* (or right-hand side). K is the *correspondence graph* (or glue graph) that relates the vertices and edges in the pattern and those in the replacement.

$$\begin{tikzcd}
P \arrow[d, "p"'] & K \arrow[l, "m"'] \arrow[r, "n"] \arrow[d, "d"] & R \arrow[d, "r"] \\
I & D \arrow[l, "m'"'] \arrow[r, "n'"] & O
\end{tikzcd}$$

Fig. 5. Graph transformation based on double-pushout

Given transformation rule $T = \langle P \xleftarrow{m} K \xrightarrow{n} R \rangle$ and input graph I, if an injective morphism $P \xrightarrow{p} I$ exists (i.e., P matches a subgraph of I), then T *is applicable to* I. If applicable, the result of applying T to I, as depicted in Fig. 5, is an output graph O, such that there exist graph D, injective morphisms $K \xrightarrow{d} D$ and $R \xrightarrow{r} O$, and morphisms $D \xrightarrow{m'} I$ and $D \xrightarrow{n'} O$, such that $\langle I, P \xrightarrow{p} I, D \xrightarrow{m'} I \rangle$ and $\langle O, R \xrightarrow{r} O, D \xrightarrow{n'} O \rangle$ are both pushouts.

2.3 Attributes

In order to transform models using graph transformation, it is necessary to categorize vertices and edges of different types. (Recall that three types of vertices and two types of edges are used in Fig. 2.) It is also necessary to take into account other attributes that further differentiate vertices, such as the ones that decide whether a port is input port or output port. Therefore, we let A be a globally defined attribute set, and extend the definition of graph G to be $\langle V_G, E_G, A_G \rangle$, where $A_G : (V_G \cup E_G) \to 2^A$ is a total function that returns a (possibly empty) set of attributes for each vertex and edge.

Our other definitions in the previous subsection remain unchanged, except that the definition of graph morphism is enhanced next to take into consideration the attributes.

2.4 Criteria Attributes and Operation Attributes

We define a subset of attributes $U \subseteq A$ to be *unchecked attributes*. It contains attributes that need not be directly checked in the extended graph morphisms to be defined below. A subset of unchecked attributes, $C \subseteq U$, is called *criteria*. Let B be an auxiliary set that equals $2^A \times 2^A$. We require any criterion $c \in C$ to be an element in 2^B. Given two vertices or edges $x \in \big(V_G \cup E_G\big)$ and $y \in \big(V_H \cup E_H\big)$, we say that criterion c is *satisfied by x matching y* if $\big(A_G(x), A_H(y)\big) \in c$.

We now extend the definition of graph morphism discussed in Sec. 2.2 to become the following. An *attributed graph morphism* from graph G to H is a graph morphism $m : V_G \to V_H$ satisfying the following additional conditions:

1. for any $v \in V_G$,
   (a) $\forall a \in \big(A_G(v) \setminus U\big).\ a \in A_H(m(v))$

(b) $\forall c \in \big(A_G(v) \cap C\big).\ \big(A_G(v), A_H(m(v))\big) \in c$

2. for any $(v_1, v_2) \in E_G$,
   (a) $\forall a \in \big(A_G((v_1, v_2)) \setminus U\big).\ a \in A_H((m(v_1), m(v_2)))$
   (b) $\forall c \in \big(A_G((v_1, v_2)) \cap C\big).\ \big(A_G((v_1, v_2)), A_H((m(v_1), m(v_2)))\big) \in c$

Case (a) of the two conditions requires that all attributes belonging to a vertex or edge in G, except the unchecked ones, be associated with the matched vertex or edge in H. Therefore, the unchecked attributes of the latter form a superset of those of the former. Case (b) of the two conditions ensures that the criteria associated any vertex or edge in G be satisfied by the matching.

Practically, for a transformation depicted in Fig. 5, only the graphs P and R contain vertices and edges with criteria attributes. In particular, we call the criteria in the replacement graph R *operations*, since they essentially enforce restrictions on the output graph that may be satisfied by performing additional attribute adding or removal operations. (In this discussion, we assume that those criteria in R can be satisfied by adding or removing attributes in the output graph O.)

Notice that because of the criteria in P and R, it may not be possible to apply transformation rule T to input graph I even if it is applicable in the sense that the pattern P matches a subgraph of I.

2.5 Basic Model Transformation

The example in Fig. 2 shows a way to represent a model with an attributed graph. In the attributed graph, three special attributes are assigned to the vertices to distinguish their types: `Actor` (visually represented by big circles), `Port` (small hollow circles) and `Relation` (filled dots). Two additional attributes identify the types of the edges: `Containment` (dashed lines) and `Connection` (solid lines). The names of the vertices are unchecked attributes that are unique at each level of the hierarchy of a transformation rule. Names at different levels may be identical.

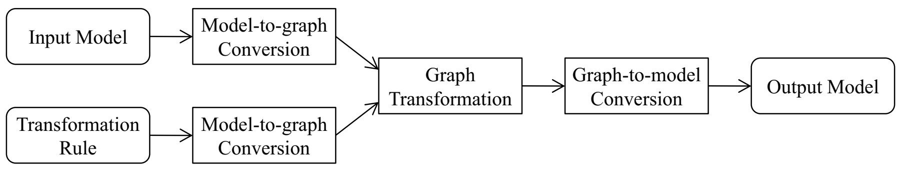

Input Model
Model-to-graph Conversion
Transformation Rule
Model-to-graph Conversion
Graph Transformation
Graph-to-model Conversion
Output Model

Fig. 6. The basic transformation process

Using the graph representation, we establish a *basic model transformation* process as shown in Fig. 6. The inputs to the process consist of an input model and a transformation rule, both specified in the modeling language. (Fig. 1 and Fig. 3 provide examples of both in this language.) The two inputs are then converted into attributed graphs. To convert a model into an attributed graph, a vertex is created for each actor, port or relation, and an edge is created for each connection or containment relation. (We use two reversed edges with the same attributes to simulate an undirected edge in Fig. 2.)

The transformation rule is converted into multiple graphs. Its pattern and replacement contain model fragments and are converted into the P graph and the R graph in Fig. 5. The correspondence table simplifies the specification of the K graph. Its "Pattern" column shows the names of the actors in the pattern. (For clearer presentation, we hide the vertices corresponding to ports and relations as well as all edges.) For a hierarchical transformation rule, the names may contain parts separated by dots, referring to the unique identifiers at different levels. The "Replacement" column shows the names of the corresponding actors in the replacement. There is a one-to-one relationship between entries in both columns. Conceptually the conversion process computes a subgraph of P as K, such that K contains only vertices listed in the "Pattern" column. The one-to-one nature ensures that a subgraph exists in R that is isomorphic to this selection of K.

After the conversion, graph transformation can be applied. The transformation result is converted back into a model for output.

3 Model-Based Transformation

Basic transformations are limited in expressiveness. To locate a match in the input model, a subgraph isomorphic problem needs to be solved, which is known to be *NP*-hard. This complexity restricts the size of patterns that can be used in practice. Furthermore, because a basic transformation is context-free, it can only operate on a subgraph matched by the pattern each time, regardless of the rest of the graph.

To enhance expressiveness, a common practice is to employ an additional mechanism to control multiple applications of basic transformations. In our work, we leverage the heterogeneous models of computation provided by Ptolemy. We allow transformation designers to create higher-order models as compositions of basic transformations. Those basic transformations in the higher-order model operate on models that are considered as first-class objects. The output of a basic transformation can be passed to the next one via its output port. The communication is governed by the model of computation used. We call such a higher-order model a *model-based transformation*, as opposed to basic transformations that do no involve any control mechanism.

3.1 TransformationRule Actor

We consider any basic transformation T as a function $F_T : \mathbb{G} \to \mathbb{G}$, where $\mathbb{G}$ is the set of all graph representations of models. For any input graph $I \in \mathbb{G}$, if T is applicable to I, F_T returns the output graph obtained by applying T to I; otherwise, F_T simply returns I. We then define the TransformationRule actor as a pure functional actor that computes F_T. It has a single input port and two output ports. The input port accepts actor tokens, which contain models to be transformed. Its first output port sends the transformation results obtained by applying F_T to the inputs. Its second output port (visually shown at the bottom of the actor's icon) produces true or false values, which signify whether the transformation was applicable for those inputs.

When the TransformationRule actor is opened in the Ptolemy II GUI (Graphical User Interface), an interface appears to allow the designer to edit the basic transformation that this actor contains. This interface has a tab for each of the three components, as is shown in Fig. 3. (For better usability, the correspondence table is automatically maintained by the tool unless the designers explicitly specify it.)

3.2 A Library of Actors for Model-Based Transformation

In addition to TransformationRule, we have created other actors in an actor library for model-based transformation.

ModelGenerator is used to generate initial actor tokens. It has different usages. If an input string containing the description of a model in the Modeling Markup Language (MoML) is provided, it parses the string and sends the model via its output port. If only a model name is provided, it creates an empty model with the given name.

ModelCombine accepts multiple input models at each time. It merges those models and outputs a combined model. Suppose the n input models are represented with graphs $G_1, G_2, \cdots, G_n$, then the output model can be represented with $G = \langle V_G, E_G, A_G \rangle$, where V_G, E_G and A_G are the disjoint unions of the vertex sets, edge sets and attribute functions (considered as sets of argument-value pairs) of the input graphs. We take an extra step after the merging to update the name attributes so that they are unique at each level of the resulting model.

ModelView displays the input models in a separate window. It updates the window when a new actor token is received. After displaying the model in the actor token, it sends the token to downstream actors via its output port.

ModelExecutor executes the input models to completion. Inputs can be provided to the models being executed via user-customized input ports of ModelExecutor. The tokens available at those input ports are automatically transmitted to the input ports with the same names of the executed models that have the same names. Outputs from the models are also transmitted to the user-customized output ports.

MoMLGenerator exports the input models in the Modeling Markup Language (MoML). The exported strings can be written into files by FileWriter.

3.3 Applications

There is a large variety of applications for model-based transformation. We sketch some of them here. A concrete example of the model construction application will be discussed in the next section.

- *Model construction.* ModelGenerator can be used to generate empty models, which are first-class objects manipulated by a higher-order model (the one that contains the ModelGenerator). Arbitrary models can thus be constructed by modifying empty models with transformations. The constructed models can be executed by ModelExecutor on the fly, or be stored in files by MoMLGenerator and FileWriter.
- *Model optimization.* When appropriate information about model behavior is provided, model transformation can be used to optimize existing models while preserving their behavior. One example is to partially evaluate a given model by eliminating the parts of computation logic that output signals that can be computed statically. The validity is based on the information about whether the actors' outputs are constant, or how actors' outputs depend on their inputs. This information may be provided in the actors' behavioral interface [10], or be obtained with a static analysis of the code that implements the actors.
- *Design refactoring.* Refactoring also preserves model behavior. It usually aims to improve model designs for better understandability or easier maintenance. Take hierarchy flattening as an example. For some models, hierarchy may be eliminated by moving actors to higher levels. An opposite operation is to introduce extra levels to the hierarchy by encapsulating actors, which helps clarify the design and protect the encapsulated parts.
- *Structural parametrization.* Models can be parametrized with placeholders defined in their structures. Model transformations can be used to configure those placeholders to form complete models. Compared to value-based parametrization, structural parametrization is a generalization that provides more design flexibility and reuse opportunity. Furthermore, one can construct a class hierarchy for models, where models at each level (except the root level) are variants obtained from structurally parametrizing some models at a higher level. Formal checking, using for example interface automata [11], can be incorporated to guarantee behavioral properties. This leads to an actor-oriented subclassing mechanism that generalizes the work in [12].
- *Execution parallelization.* Using a model of computation that provides concurrency, multiple models can be executed in parallel with ModelExecutors. Those models can communicate with each other via the user-customized input ports and output ports of the ModelExecutors. This makes it possible to simulate a distributed system, where the models are executed on separate computers and communicate with each other.
- *Workflow automation.* Model-based transformation can also be used to automate tasks in the model development workflow. Those tasks include component configuration and composition, version control, and regression testing.

4 MapReduce Example

In this section we discuss a parametrized higher-order model that generates a MapReduce model and executes it. Fig. 7 shows part of the hierarchy of this

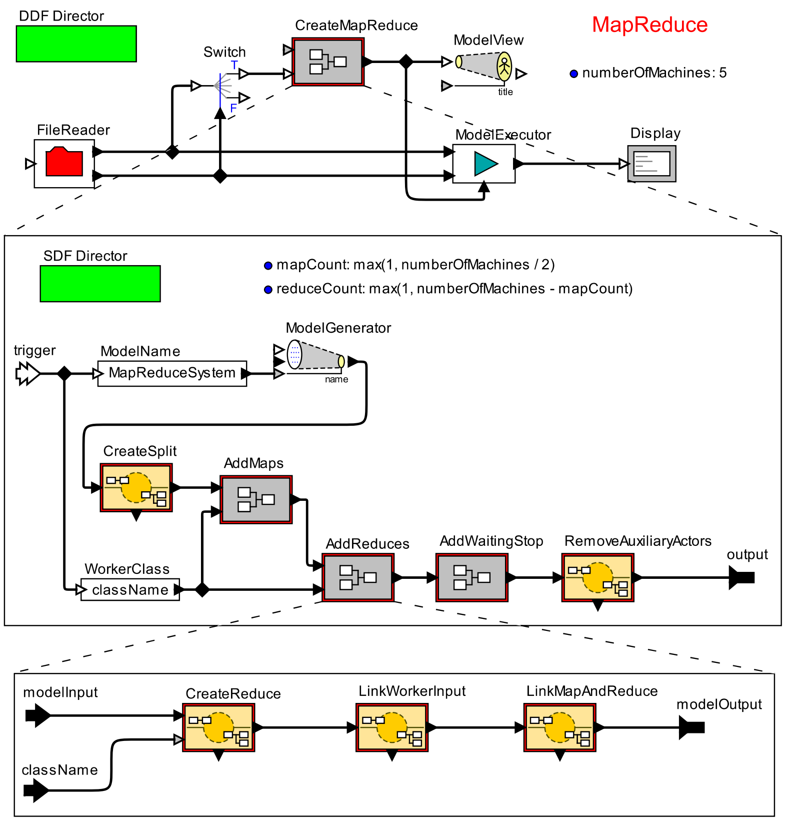
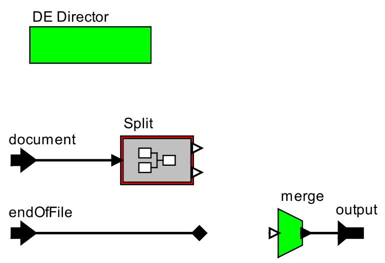


Fig. 7. A higher-order model that generates a MapReduce model and executes it

higher-order model. A parameter at the top level, `numberOfMachines`, defines the preferred number of worker machines, which can be set by the user. For simulation, we use a FileReader to read the documents from disk and to output them in tokens via the upper output port. The lower output port produces true when all documents are sent, or false otherwise. When it outputs false, the document tokens are queued in the buffer for the ModelExecutor's upper-left input port. When it outputs true, the Switch sends a token to the CreateMapReduce actor, triggering it to generate a MapReduce model in an actor token. (Fig. 1 shows the MapReduce model generated when `numberOfMachines` equals 5.) The model is then sent to the ModelExecutor for immediate execution. The buffered documents are provided to the model as inputs at its "document" input port, and its outputs to the "result" output port are automatically transmitted to the Display actor connected to the ModelExecutor's output port.



(a) Pattern (empty) (b) Replacement

	Pattern	Replacement
1	(empty)	

(c) Correspondence

Fig. 8. The transformation rule in CreateSplit

The CreateMapReduce actor contains an SDF (Synchronous Dataflow) model that controls several TransformationRule actors (each represented with a yolk-like icon surrounded by a box). `mapCount` is defined to be the number of Map machines, and `reduceCount` is the number of Reduce machines. Fig. 8 shows the transformation rules in the CreateSplit actor. It creates a Split actor and a Merge actor, together with the input ports and output port of the MapReduce model. Its pattern is deliberately made empty so that the transformation rule is applicable to the empty model generated by the ModelGenerator. In Fig. 3, we have shown the transformation rule in the LinkMapAndReduce actor. It connects the ports of an arbitrary Map actor and an arbitrary Reduce actor, with a constraint to make sure that no duplicated connections are made in multiple applications of the same transformation rule. We set a special parameter, which is not shown in the interface, so that the transformation is applied exactly (`mapCount` × `reduceCount`) times for each input model, so that all the Map actors and Reduce actors in it are interconnected.

This example shows how we use model transformation as a tool to construct a complex model. The size of the dynamically generated model is parametrizable with an integer parameter, and has no impact on the size of the higher-order model that the designer manually creates. Each TransformationRule actor can be separately designed, documented and maintained. It can also be parametrized and reused to construct other models.

5 Related Work

Model transformation has been under active research in recent years. In recognition of the public interest, the OMG (Object Management Group) has issued a request for proposal (RFP) on MOF (Meta-Object Facility) QVT (Query /

Views / Transformations) to seek a standardized approach to model transformation [13].

Besides our model transformation tool developed in the Ptolemy II framework, existing tools include AGG [14], PROGRES [15], AToM^3 [16], FUJABA [17], VIATRA2 [18], and GReAT [19]. Among those, our tool is the only one that supports a large and extensible collection of MoCs for controlling basic transformations. By carefully choosing MoCs, sequential transformation and parallel transformation can be achieved, as well as a mixture of both. This control mechanism is more flexible than the priority-based control provided by AGG and AToM^3, the imperative program control implemented in PROGRES, and the restricted selection of MoCs that the other tools offer. Furthermore, our model transformation tool provides a user-friendly language for transformation specification. It employs the same visual language as that used for manually creating models. This frees designers from learning another language for specifying transformations (such as a textual language and UML class diagrams), understanding the meta-models of their models, and describing their transformations in terms of the meta-models. Higher-order model composition for embedded system design is proposed in [20] and [21]. Compared to other related approaches in this field, such as Ptalon [4] and higher-order Petri nets [22], our model-based transformation approach allows designers to visually describe pieces of model structures and to transform them step by step. Besides, our model descriptions are themselves hierarchical heterogeneous models, which can be divided into parametrized components for reuse. Therefore, not only the models constructed by the descriptions can easily scale to large sizes, the descriptions themselves are also scalable.

6 Conclusion

We present our approach to higher-order model composition based on model transformation. We provide a formal definition of graph transformation, which serves as the basis of our model transformation technique. We show that basic transformations can be used as actors in a hierarchical heterogeneous models. Our approach makes it easy to create complex transformations as composition of basic ones. We provide a word-counting model designed using the MapReduce programming pattern as a concrete example.

References

1. Goderis, A., Brooks, C., Altintas, I., Lee, E.A., Goble, C.A.: Composing different models of computation in Kepler and Ptolemy II. In: International Conference on Computational Science (ICCS), Beijing, China (2007) 182–190
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Operating Systems Design and Implementation (OSDI), San Francisco, California, USA (2004) 137–150
3. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. Proceedings of the IEEE **91**(2) (2003) 127–144

4. Cataldo, A., Cheong, E., Feng, T.H., Lee, E.A., Mihal, A.C.: A formalism for higher-order composition languages that satisfies the Church-Rosser property. Technical Report UCB/EECS-2006-48, EECS Department, University of California, Berkeley (2006)
5. Cataldo, J.A.: The Power of Higher-Order Composition Languages in System Design. PhD thesis, EECS Department, University of California, Berkeley (2006)
6. Königs, A.: Model transformation with triple graph grammars. In: Model Transformations in Practice Workshop. (2005)
7. Schürr, A.: Specification of graph translators with triple graph grammars. In: WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Springer-Verlag (1994) 151–163
8. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: Annual Symposium on Foundations of Computer Science (FOCS). (1973) 167–180
9. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Mathematical. Structures in Comp. Sci. **11**(5) (2001) 637–688
10. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. Formal Aspects of Computing **16**(3) (2004) 210–237
11. de Alfaro, L., Henzinger, T.A.: Interface automata. ACM SIGSOFT Software Engineering Notes **26**(5) (2001) 109–120
12. Lee, E.A., Liu, X., Neuendorffer, S.: Classes and inheritance in actor-oriented design. ACM Transactions on Embedded Computing Systems (TECS) **to appear** (2008)
13. Object Management Group (OMG): Request for proposal: MOF 2.0 Query / Views / Transformations RFP (2002)
14. Taentzer, G.: AGG: A tool enviroment for algebraic graph transformation. In: Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE), Kerkrade, The Netherlands (1999)
15. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: Proceedings of the 5th European Software Engineering Conference, Sitges, Spain (1995) 219–234
16. Lara, J.d., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, Grenoble, France (2002)
17. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: ICSE '00: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (2000)
18. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, Esslingen, Germany (2006) 1280–1287
19. Agrawal, A., Karsai, G., Shi, F.: A UML-based graph transformation approach for implementing domain-specific model transformations. International Journal on Software and Systems Modeling (2003)
20. Reekie, H.J.: Realtime Signal Processing – Dataflow, Visual, and Functional Programming. PhD thesis, University of Technology at Sydney (1995)
21. Colaço, J.L., Girault, A., Hamon, G., Pouzet, M.: Towards a higher-order synchronous data-flow language. In: EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software, New York, NY, USA, ACM Press (2004) 230–239
22. Janneck, J.W., Esser, R.: Higher-order Petri net modeling – techniques and applications. In: Workshop on Software Engineering and Formal Methods. (2002)

ISE language: the ADL for Efficient Development of Cross Toolkits

Nikolay Pakulin and Vladimir Rubanov

Institute for System Programming of the Russian Academy of Sciences,
Moscow, Russia,
npak@ispras.ru vrub@ispras.ru

Abstract. Cross toolkits (assembler, linker, debugger, simulator, profiler) are widely used for software-hardware codesign; an early creation of cross toolkits is an important success factor for industrial embedded systems. At the hardware design stage systems are subject to significant design alterations including changes in the instruction set of target CPUs. This is a challenging issue for early cross toolkit development. In this paper, we present a new Architecture Description Language (ADL) called *ISE language* and an approach to early cross toolkit development to cope with hardware design changes. The paper introduces the MetaDSP framework that supports ISE-based construction of cross toolkits and gives brief overview of the MetaDSP applications to industrial projects that proves the industrial strength of the presented approach and tools.

1 Introduction

Nowadays we witness creation of various embedded systems with rather strict constraints (chip size, power consumption, performance) not only for aerospace and military applications but also for industry and even consumer electronics. The constant trend of cost and schedule reduction in microelectronics hardware design and development makes it reasonable to develop special-purpose computing systems for various applications and gives new impulse to the market of embedded systems. Such systems consist of a dedicated hardware platform developed for a particular application and a problem-specific software optimized for that hardware.

Cross tools play an important role for bringing an embedded system to life as they allow development, debugging and profiling of the target software on powerful workstations which do not suffer from the limitations of the target embedded systems and typically run on CPUs which architecture and instruction set are different from the target CPUs. Primary components of such cross toolkits are assembler, linker, simulator, debugger, and profiler. Unlike chip production, development of cross toolkits does not require precise hardware design description; it is sufficient to have just high-level definition of the target hardware platform: the memory/register architecture and the instruction set with cycle specification. This allows developing cross tools as soon as the early design stages even if exact VHDL/Verilog specification is not ready yet. Such co-development has the following crucial benefits:

- Hardware prototyping and design space exploration (e.g. [1] and [2]) – early development, execution and profiling of sample programs allows study and estimation of the overall design adequacy as well as efficiency of particular design ideas such as adding/removing instructions, functional blocks, registers or whole co-processors.
- Early software development including development, debugging and optimizing the software *before* the target hardware production. It reduces time-to-market for the complete "Hardware + Software" product.
- Hardware design validation. The developed cross-simulator could be used to run test programs against VHDL/Verilog-based simulators. This capability could not be overestimated for the quality assurance before actual silicon production.

The first feature mentioned above – design space exploration – results in frequent changes of requirements. System designers may decide to add or remove an instruction or modify the register file of the CPU. Cross toolkit developers must rapidly answer to such changes and produce new version of the toolkit in short terms. Besides, this practice imposes certain quality and performance requirements on the cross toolkits and on the simulator in particular. Special attention should be paid to the performance efficiency of the simulator.

1.1 Related Work

Efficient cross toolkit development process requires automation to minimize time and effort necessary to update the toolkit to match new requirements. Such automation is built around a machine-readable definition of the target hardware platform. There are three groups of languages suitable for this task:

- Hardware Definition Languages (HDL, [3]) used for detailed definition of the hardware;
- Architecture Description Languages (ADL, [4] and [5]) used for high-level description of the hardware;
- and general purpose programming languages (such as C/C++).

HDL specifications define CPU operations with very high level of detail. All three major modern HDL – VHDL [6], Verilog [7], and SystemC [8] – have execution environments that can serve as a simulator to run any assembly language programs for the target CPU: Synopsys VCS, Mentor Graphics ModelSim, Cadence NC-Sim and other. Still, low performance of HDL-based simulators is one of the major obstacles for HDL application in cross toolkit development. Another issue is the late moment of HDL description availability: it appears after completing the instruction set design and functional decomposition. Furthermore, HDL does not contain an explicit instruction set definition that makes automated assembler/disassembler development impossible. These issues prevent from using HDL to automate cross toolkit development.

Architecture Description Languages (such as nML[9], ISDL[10], EXPRESSION[11]) are under active development during the recent decade. There are tools created for rapid hardware prototyping at the high level including cross toolkit generation. Corresponding approaches are really good for early design phase since they help to explore key design decisions. Unfortunately, at the later design stages details in an ADL description become smaller, the size of the description grows and sooner or later it comes across the limitations of the language. As a result, is breaks the efficiency of the simulator generated from the ADL description and makes the profiler to give only rough performance estimates without clear picture of bottlenecks. Cross toolkits completely generated from an ADL description are not applicable for industrial-grade software development yet.

Manual coding with C or C++ language gives full control over all possible details and allows creation of cross toolkits of industrial quality and efficiency. Many companies offer services on cross toolkit development in C/C++ (e.g. TASKING, Raisonance, Signum Systems, ICE Technology, etc.). Still it requires significant efforts and (what is more important) time to develop the toolkit from scratch and maintain it aligned with the requirements. Long development cycle makes it almost impossible to use cross toolkits developed in C/C++ for hardware prototyping and design space exploration.

1.2 Paper Overview

In this paper, we present a new approach to cross toolkit development that combines the power of high-level definition using ADL-like language and the efficiency of the modern programming languages. The method provides reasonable level of automation with support for rapid requirement changes and co-development of hardware and software components of modern embedded systems.

The article is organized as follows. Section 2 presents the new ADL language for defining instruction set called *ISE*. Section 3 introduces MetaDSP framework for cross toolkit development that uses hybrid hardware description with both high-level ADL part and efficient C/C++ part. Section 4 briefly overviews several industrial applications of ISE and MetaDSP framework. Conclusion summarizes the lessons learned and gives some perspectives for future development.

2 ISE Language

We developed ISE (*I*nstruction *S*et *E*xtension) language to specify hardware design elements that are subject to most frequent changes: memory architecture and CPI instruction set. ISE description is used to generate assembler and disassembler tools completely and to generate components of the linker, debugger and simulator tool.

The following considerations guided the language design:

– the structure of an ISE description should follow the typical structure of an instruction set reference manual (like [12] or [13]) that usually serve as the input for the ISE description development;

- support for irregular encoding of instructions typical for embedded DSP applications including support for large number of various formats, distributed encoding of operands in the word, etc.;
- operational definition of data types, logic and arithmetic instructions, other executable entities should be specified in a C-like programming language.

ISE module consists of 7 sections:

1. **.architecture** defines global CPU architecture properties such as pipeline stages, CPU resources (buses, ALUs, etc.), initial CPU state;
2. **.storage** defines memory structure including memory ranges, I/O ports, access time;
3. **.ttypes** and **.otypes** define data type to represent registers and instruction operands;
4. **.instructions** defines CPU instruction set (see 2.1);
5. **.aspects** defines various aspects of binary encoding of CPU instructions or specifies additional resources or operational semantics of instructions;
6. **.conflicts** specifies constraints on sequential execution of instructions such as potential write after read register or bus conflict; assembler uses conflict constraints to automatically insert NOP instructions to prevent conflicts during software execution.

2.1 Instruction Definition

.instruction section is the primary section an ISE module. It defines the instruction set of the target CPU. For each instruction cross toolkit developers can specify:

- mnemonics and binary encoding;
- reference manual entry;
- instruction properties and resources used;
- instruction constraints and inter-instruction dependencies;
- definition of execution pipeline stage.

Mnemonics part of an instruction definition is a template string that specifies fixed part of mnemonics (e.g. `ADD`, `MOV`), optional suffixes (e.g. `ADDC` or `ADDS`) and operands. A singe instruction might have several definitions depending on the operand types. For example, `MOV` instruction could have different definitions for register-register operation, register-memory and memory-memory operations.

Binary encoding is a template that specifies how to encode/decode instructions depending on the instruction name, suffixes and operands.

Reference manual entry is a human-readable specification of the instruction.

Properties and resources specify external aspects of the instruction execution such as registers that it reads and writes, buses that the instruction accesses, flags set etc. This information is used to detect and resolve conflicts by the assembler tool. Besides this the instruction definition might specify explicit dependencies on preceding or succeeding instructions in the constraints and dependencies section.

ISE language contains an extension of C programming language called ISE-C. This extension is used to specify execution of the operation on each pipeline stage. ISE-C has extra types for integer and fixed point arithmetic of various bit length, new built-in bit operators (e.g. shift with rotation), built-in primitives for bit handling. ISE-C has some grammar extension for handling operands and optional suffixes in mnemonics. Furthermore ISE-C expression can use a large number of functions implemented in ISE core library.

An example of instruction specification is presented at figure 1.

```
/*
 * This is a C-style block comment.
 */
// This is a C++-style one-line comment.
// <ALU001> - the identifier of the definition.
// ADD[S:A][C:B]  - instruction mnemonics with optional parts.
//  Actually defines 4 instructions: ADD, ADDS, ADDC, ADDSC.
//  GRs, GRt - identifiers of a general-purpose register.
//   Rules for binary encoding of GRs and GRt are defined in
//   .otypes section.
<ALU001>  ADD[S:A][C:B] {GRs}, {GRt}
    // Binary encoding rule.
    // For example, "ADDC R0, R1" is encoded as
    // 0111-0001-1000-1001
    0111-0A0B-1SSS-1TTT
    // The reference manual string.
    "ADD[S][C] GRs, GRt"
    // instruction properties:
    //  reads the registers GRs and GRt,
    //  writes the register GRs.
    properties [ wgrn:GRs, rgrn:GRs, rgrn:GRt ]
    // Operation of the EXE pipeline stage
    // specifies using ISE-C language.
    action {
      alu_temp = GRs + GRt;
      // If the suffix ‘C’ is set in mnemonics
      // use ‘getFlag’ function from the core library.
      if (#B) alu_temp += getFlag(AC0);
      // If the suffix ‘S’ is set in mnemonics
      // use ‘SAT16’ function from the core library.
      if (#A) alu_temp = SAT16(alu_temp);
      GRs = alu_temp;
    }
```

Fig. 1. An example of instruction specification.

Please note that unlike classic ADL languages ISE specification does not provide the complete CPU model. The purpose of ISE is to simplify definition of the elements that are subject to the most frequent changes. All the rest of the model is specified using C/C++ code. This separation allows for flexible and maintainable hardware definition along with high performance and cycle-precise simulation.

3 Development Process

The proposed hybrid ADL/C++ hardware definition is supported by the *MetaDSP* framework for cross-toolkit development. The framework includes:

– ISE translator that generates components of cross tools from the ISE specification;
– pre-defined components for ISE development (e.g. ISE-C core functions library);
– an IDE for hardware definition development (in ISE and C++), target software development (in Embedded C[14] and assembly languages), controlled execution within simulator; the Embedded C compiler supports a number of optimizations specific for DSP applications[15].

Figure 2 presents the structure of the MetaDSP framework.

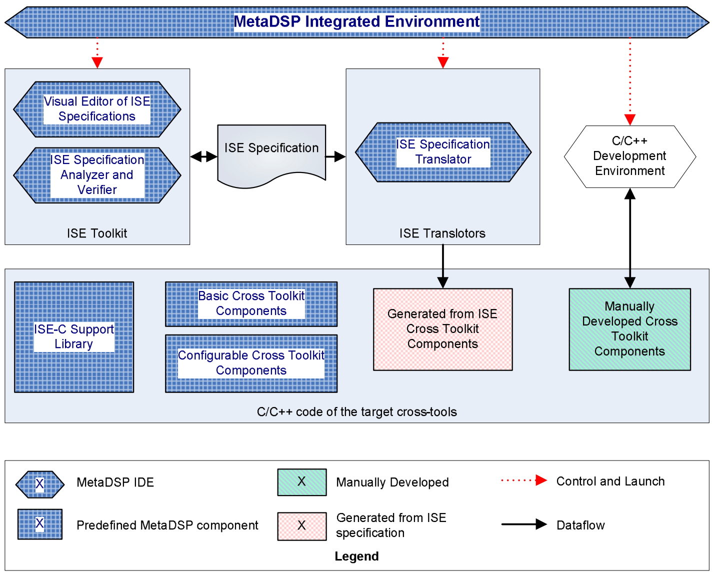

MetaDSP Integrated Environment
Visual Editor of ISE Specifications
ISE Specification Analyzer and Verifier
ISE Toolkit
ISE Specification
ISE Specification Translator
ISE Translotors
C/C++ Development Environment
ISE-C Support Library
Basic Cross Toolkit Components
Configurable Cross Toolkit Components
Generated from ISE Cross Toolkit Components
Manually Developed Cross Toolkit Components
C/C++ code of the target cross-tools
MetaDSP IDE
Predefined MetaDSP component
Manually Developed
Generated from ISE specification
Control and Launch
Dataflow
Legend

Fig. 2. MetaDSP framework structure

MetaDSP toolkit uses ISE specification to generate cross tools and components. For example, the MetaDSP tools generate assembler and disassembler tools completely from the ISE specification. For linker MetaDSP generates information about instruction binary encodings, instruction operands and relocatable instructions. Debugger and profiler use memory structures and operand types from the ISE specification.

The cycle-precise simulator is an important part of the toolkit. Figure 3 presents its architecture. MetaDSP tools generate several components from the ISE specification: memory implementation (from `.storage` section), resources (from `.architecture` section), instruction implementations and decoding tables (from `.instruction` section), as well as conflicts detector and instruction metadata.

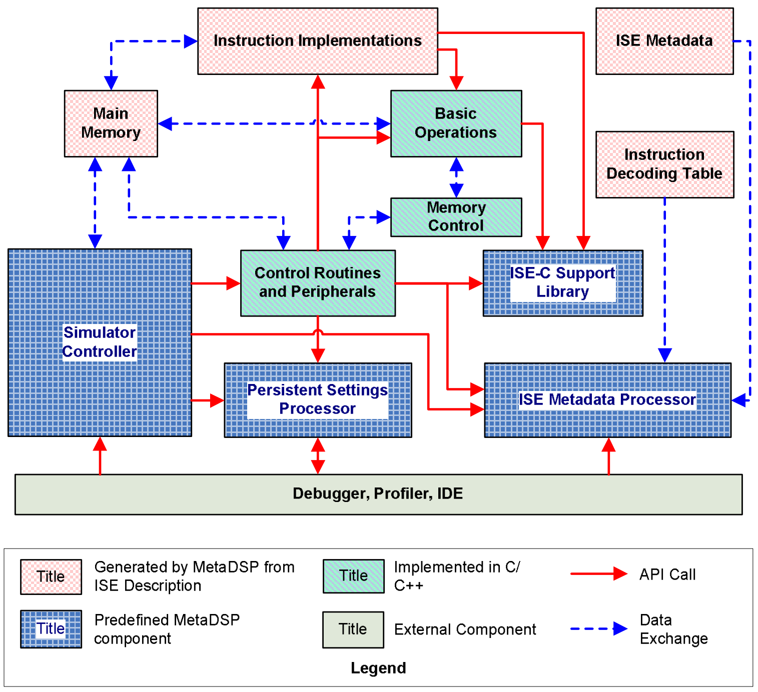


Fig. 3. MetaDSP simulator architecture

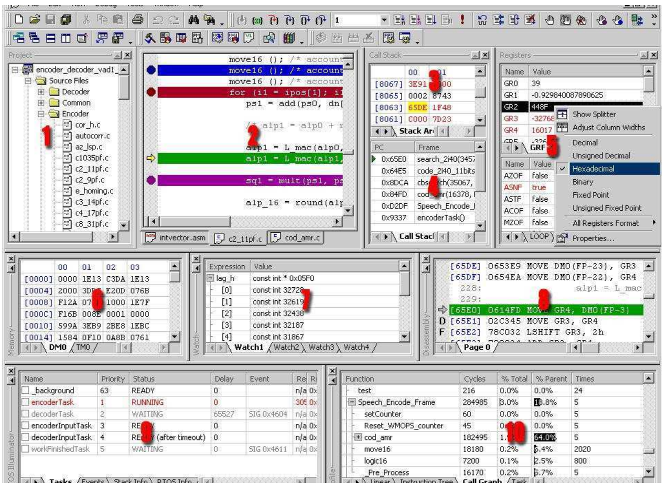
Within the presented approach certain components are specified in C++:

– control logic, including pipeline control (if any), address generation, instruction decoder;
– memory control;
– model of the peripheral devices including I/O ports.

For most of the manual components MetaDSP tools generate stubs or some basic implementation in C++. Developers may use the generated code to implement peculiarities of the target CPU, such as jumps prediction, instruction reordering, etc.

Using C/C++ to implement CPU control logic and memory model facilitates high performance of the simulator. Another benefit of using C/C++ compared to true ADL languages is an early development of the cross toolkit: it might start before completing the function decomposition of the target CPU; thus the simulator could be used to experiment with design variations.

Figure 4 presents the snapshot of OSCAR Studio, the IDE for MetaDSP framework.



```
              move16 (); /* account
              move16 (); /* account
              move16 (); /* account
              for (i1 = ipos[1]; i1
                  ps1 = add(ps0, dn[

                  /* alp1 = alp0 + r

                  alp1 = L_mac(alp0,
                  alp1 = L_mac(alp1,

                  sq1 = mult(ps1, ps

                  alp_16 = round(alp
```

	00	01
[8067]	3E91	00
[8065]	0002	8743
[8063]	65DE	1F48
[8061]	C000	7D23

PC	Frame
0x65E0	search_2i40(3457
0x64E5	code_2i40_11bits
0x8DCA	cbs ch(35067,
0x84FD	cod mr(16378,
0xD2DF	Speech_Encode_
0x9337	encoderTask()

Name	Value
GR0	39
GR1	-0.929840087890625
GR2	448F
GR3	-32768
GR4	16017

Name	Value
AZOF	false
ASNF	true
ASTF	false
ACOF	false
MZOF	false

	00	01	02	03
[0000]	0000	1E13	C3DA	1E13
[0004]	2000	3D	E20D	076B
[0008]	F12A	07	1000	1E7F
[000C]	F16B	008E	0001	0000
[0010]	599A	3EB9	2BE8	1EBC
[0014]	1584	0F10	0A8B	0761

Expression	Value
lag_h	const int * 0x05F0
[0]	const int 32728
[1]	const int 32619
[2]	const int 32438
[3]	const int 32187
[4]	const int 31867

```
   [65DE] 0653E9 MOVE DM0(FP-23), GR3
   [65DF] 0654EA MOVE DM0(FP-22), GR4
     228:                        alp1 = L_mac
     229:
   [65E0] 0614FD MOVE GR4, DM0(FP-3)
 D [65E1] 02C345 MOVE GR3, GR4
 F [65E2] 78C032 LSHIFT GR3, 2h
```

Name	Priority	Status	Delay	Event	Re	R
_background	63	READY	0		n/a	0x
encoderTask	1	RUNNING	0		305	0x
decoderTask	2	WAITING	65527	SIG 0x4604	n/a	0x
encoderInputTask	3	RE	0		n/a	0x
decoderInputTask	4	RE (after timeout)	0		n/a	0x
workFinishedTask	5	WAITING	0	SIG 0x4611	n/a	0x

Function	Cycles	% Total	% Parent	Times
test	216	0.0%	0.0%	24
Speech_Encode_Frame	284985	3.0%	8.8%	5
setCounter	60	0.0%	0.0%	5
Reset_WMOPS_counter	45	0	0.0%	5
cod_amr	182495	1.	64.0%	5
move16	18180	0.2%	6.4%	2020
logic16	7200	0.1%	2.5%	800
_Pre_Process	16170	0.2%	5.7%	5

Fig. 4. OSCAR Studio: the IDE for MetaDSP framework

Red numbers mark various windows of the IDE:

1. Project Navigator window. It displays the tree of the source files and data files.
2. Source Code Editor window. The editor supports syntax highlight and instruction autocompletion (from the ISE specification). The editor window is integrated with the debugger - it marks break points, frame count points and trace points.
3. Stack Memory window displays the contents of the stack.
4. Call Stack window displays the enclosing frames (both assembly subroutines and C functions).
5. Register window displays the contents of the CPU registers.
6. Memory dump window displays contents of various memory regions.

7. Watch window displays the current value of arbitrary C expressions.
8. Code Memory window displays the instructions being executed. It supports both binary and disassembly forms as well as displaying the current pipeline stage (fetch, decode, execute, etc.).
9. OS debugger window displays the current state of the execution environment (OS): list of the current tasks, semaphores, mutexes, etc.
10. Profiler window displays various profiling data. The profiler is integrated with the editor window as well – the editor can show profiling information associated with code elements.

4 Industrial Applications

The approach presented in this paper and MetaDSP framework were applied to five industrial projects. Please note that the each "major releases of the cross toolkit" mentioned in the project list below is caused by a major change in CPU design such as modification of the instruction set or memory model alteration.

– 16-bit RISC DSP CPU with fixed point arithmetic. Produced 25 major releases of the cross-toolkit.
– 16-bit RISC DSP CPU with support for Adaptive Multi-Rate (AMR) sound compression algorithm. Produced 25 major releases of the cross-toolkit.
– 32-bit RISC DSP CPU with support for Fourier transform and other DSP extensions. Produced 39 major releases of the cross-toolkit.
– 16/32-bit RISC CPU clone of ARM9 architecture.
– 16/32-bit VLIW DSP CPU with support for Fourier transform, DMA, etc. Produced 33 major releases of the cross-toolkit.

The following list summarizes lessons learned from the practical applications of the approach. We compared time and effort needed in a pure C++ development cycle of cross toolkits with the ISE-enabled process:

– size of assembler, disassembler and simulator sources (excluding generated code), in lines of code: reduced by 12 times;
– cross-toolkit development team (excluding C compiler development): reducing from 10 to 3 engineers;
– number of errors detected in the presentation of hardware specifications in cross tools: reduction by the factor of more than 10;
– average duration of the toolkit update: reduced from several days to hours (even minutes in many cases).

4.1 Performance Study

This section presents a performance study of a production implementation of the AMR sound compression algorithm. The study was performed on Intel Core 2 Duo 2.4 GHz.

The size of the implementation was 119 C source files and 142 C header files, and 25 files in the assembly language; total size of sources was 20.2 thousand

LOC without comments and empty lines. The duration of the audio sample (10 seconds voice speech) lasted 670 million of cycles on the target hardware.

Table 1 presents elapsed time measurements of the generated cross tools for the AMR case study. Table 2 presents measurements of the generated simulator in MCPS (millions of cycles per second).

Table 1. AMR sample – cross toolkit performance

Operation	Duration, sec.
Translation (.c → .asm)	22
Assembly (.asm → .obj)	14
Link (.obj → .exe)	1
Build, total	**37**
Execution on the audio sample (fast mode)	53
Execution on the audio sample (debug mode with profiling)	93

Table 2. AMR sample – simulator performance

Execution mode	MCPS
Fast mode	12.6
Debug mode with profiling	7.2
Peak performance on a synthetic sample	25.0

5 Conclusion

The paper presents an approach to automation of cross toolkit development for special-purpose embedded systems such as DSP and microcontrollers. The approach aims at creation the cross tools, namely assembler/disassembler, linker, simulator, debugger, and profiler, at early stages of system design. Early creation of the cross tools gives opportunity to prototype and estimate efficiency of design variations, co-development of the hardware and software components of the target embedded system, and verification and QA of the hardware specifications before silicon production.

The presented approach relies on a two-level description of the target hardware: description of the most flexible part – the instruction set and memory model – using the new ADL language called *ISE* and description of complex fine grained functional aspects of CPU operations using a general purpose programming language (C/C++). Having ADL descriptions along with a framework to generate components of the target cross toolkits and common libraries brings high level of responsiveness to frequent changes in the initial design that are a common issue for modern industrial projects. Using C/C++ gives cycle-accurate simulation and overall efficiency of the cross toolkits that meets the needs of industrial developers. The approach is supported by a family of tools comprising MetaDSP framework.

The approach is applicable to various embedded systems with RISC core architectures. It supports simple pipelines with fixed number of stages, multiple memory banks, instructions with fixed and variable cycle count. These facilities cover most of modern special purpose CPUs (esp. DSP) and embedded systems. Still some features of modern general purpose high performance processors lay beyond the capabilities of the presented approach: superscalar architectures, microcode, instruction multi-issue, out-of-order execution. Besides this, the basic memory model implemented in MetaDSP does not support caches, speculative access, etc.

Despite the limitations of the approach mentioned above it was successfully applied in a number of industrial projects including 16 and 32-bit RISC DSPs and 16/32 ARM CPUs. Number of major design changes (with corresponding releases of cross toolkits) ranged in those projects from 25 to 40. The industrial applications of the presented approach proved the concept of using the hybrid ADL/C++ description for automated development of cross toolkits in a volatile design process.

References

1. M. Hartoog, J. Rowson, P. Reddy. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. Design Automation Conference (DAC) 1997.
2. Lin Yung-Chia. Hardware/Software Co-design with Architecture Description Language. Programming Language Lab. NTHU. 2003.
3. Z. Navabi. Languages for Design and Implementation of Hardware. The VLSI Handbook2nd ed. CRC Press, 2007.
4. P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. IEEE Proceedings Computers and Digital Techniques. Vol. 152, No. 3, May 2005.
5. H. Tomiyama, A. Halambi, P. Grun, N. Dutt, A. Nicolau. Architecture Description Languages for Systems-on-Chip Design. Proc. Asia Pacific Conf. on Chip Design Language, 1999, pp. 109116.
6. VHDL Language Reference Manual. IEEE Std 1076-1987.
7. Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Std 1364-2005.
8. System C Language Reference Manual. IEEE Std 1666-2005.
9. A. Fauth, J. Van Praet, M. Freericks. Describing instruction set processors using nML. In Proc. of EDTC, 1995.

10. G. Hadjiyannis, S. Hanono, S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Design Automation Conference (DAC) 1997.
11. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. DATE 99.
12. MicroDSP 2 Instruction Set Description. VIA Technologies Manual, 2005.
13. TMS320C6000 CPU and Instruction Set Reference Guide. Texas Instruments Literature Number SPRU189F. http://focus.ti.com/lit/ug/spru189g/spru189g.pdf.
14. ISO/IEC TR 18037:2008. Programming languages – C – Extensions to support embedded processors. 2004.
15. V. Rubanov, A. Grinevich, D. Markovtsev. Programming and Computing Software Vol. 32-1, pp. 19-30, 2006

Towards Model-Based Integration of Tools and Techniques for Embedded Control System Design, Verification, and Implementation

Joseph Porter, Gábor Karsai, Péter Völgyesi, Harmon Nine, Peter Humke, Graham Hemingway, Ryan Thibodeaux, and János Sztipanovits

Institute for Software Integrated Systems,
Vanderbilt University,
Nashville TN 37203, USA,
`jporter@isis.vanderbilt.edu`,
WWW home page: `http://www.isis.vanderbilt.edu`

Abstract. While design automation for hardware systems is quite advanced, this is not the case for practical embedded systems. The current state-of-the-art is to use a software modeling environment and integrated development environment for code development and debugging, but these rarely include the sort of automatic synthesis and verification capabilities available in the VLSI domain. We present a model-based integration environment which uses a graphical architecture description language (EsMoL) to pull together control design, code and configuration generation, platform-specific resimulation, and a number of other features useful for taming the heterogeneity inherent in safety-critical embedded control system designs. We describe concepts, elements, and development status for this suite of tools.

1 Introduction

Embedded software often operates in environments critical to human life and subject to our direct expectations. We assume that a handheld MP3 player will perform reliably, or that the unseen aircraft control system aboard our flight will function safely and correctly. Safety-critical embedded environments require far more care than provided by the current best practices in software development. Embedded systems design challenges are well-documented [1], but industrial practice still falls short of these expectations for many kinds of embedded systems.

In modern designs, graphical modeling and simulation tools (e.g. Mathworks' Simulink/Stateflow) represent physical systems and engineering designs using block diagram notations. Design work revolves around simulation and test cases, with code generated from "‘complete"’ designs. Control designs often ignore software design constraints and issues arising from embedded platform choices. At early stages of the design, platforms may be vaguely specified to engineers as sets of tradeoffs.

Software development uses UML (or similar) tools to capture concepts such as components, interactions, timing, fault handling, and deployment. Workflows focus on source code organization and management, followed by testing and debugging on target hardware. Physical and environmental constraints are not represented by the tools. At best such constraints may be provided as documentation to developers.

Complete systems rely on both aspects of a design. Designers lack tools to model the interactions between the hardware, software, and the environment. For example, software generated from a carefully simulated functional dataflow model may fail to perform correctly when its functions are distributed over a shared network of processing nodes. Cost considerations may force the selection of platform hardware that limits timing accuracy. Neither aspect of development supports comprehensive validation of certification requirements to meet government safety standards.

We propose a suite of tools that aim to address many of these challenges. Currently under development at Vanderbilt's Institute for Software Integrated Systems (ISIS), these tools use the Embedded Systems Modeling Language (ESMoL), which is a suite of domain-specific modeling languages (DSML) to integrate the disparate aspects of a safety-critical embedded systems design and maintain proper separation of concerns between engineering and software development teams. Many of the concepts and features presented here also exist separately in other tools. We describe a model-based approach to building a unified model-based design and integration tool suite which has the potential to go far beyond the state of the art.

In the sequel we will provide an overview of the tool vision, and then describe the features of these tools from the point of view of available functionality. Note that two different development processes will be discussed – the development of a distributed control system implementation (by an imagined user of the tools), and our development of the tool suite itself. The initial vision section illustrates how the tools would be used to model and develop a control system. The final sections describe different parts of our tool-development process in decreasing order of maturity. We strive for clarity, with an apology to the diligent reader where the distinction is unclear.

2 Toolchain Vision and Overview

In this work, we envision a sophisticated, end-to-end toolchain that supports not only construction but also the verification of the engineering artifacts (including software) for high-confidence applications. The development flow provided by the toolchain shall follow a variation of the classical V-model (with software and hardware development on the two branches), with some refinements added at the various stages. Fig. 1 illustrates this development flow.

Consider the general class of control system designs for use in a flight control system. Sensors, actuators, and data networks are designed redundantly to mitigate faults. The underlying hardware implements a variant of the time-triggered

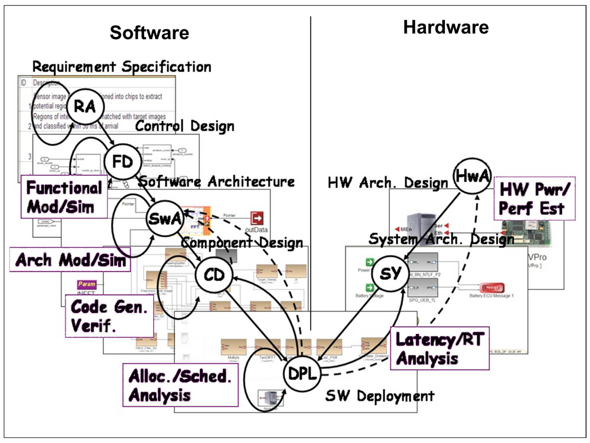


Fig. 1. Conceptual model of the toolchain: Development flow

architecture (TTA) [2], which provides precise timing and reliability guarantees. Safety-critical tasks and messages execute according to strict precomputed schedules to ensure synchronization between replicated components and provide fault mitigation and management. Software implementations of the control functions must pass strict certification requirements which impose constraints on the software as well as on the development process.

A modeling language to support this development flow must have several desired properties: (1) the ability to capture the relevant aspects of the system architecture and hardware, (2) ability to "understand" (and import) functional models from existing design tools, (3) support for componentization of functional models, and (4) ability to model the deployment of the software architecture onto the hardware architecture. The ability to import existing models from functional modeling tools is not a deeply justified requirement, it is merely pragmatic. EsMoL provides modeling concepts and capabilities that are highly compatible with AADL [3]. The chief differences are that EsMoL aims for a simpler graphical entry language, a wider range of execution semantics, and most important model-enabled integration to external tools as described below. Model exchange with AADL tools may be desirable in the future. A simple sample design will introduce key points of our model-based development flow and illustrate language concepts.

Our language design was influenced by two factors: (1) the MoC implemented by the platform and (2) the need for integration with legacy modeling and embedded systems tools. We have chosen Simulink/Stateflow as the supported "legacy" tool. As our chosen MoC relies on periodically scheduled time-triggered components, it was natural to use this concept as the basis for our modeling language and interpret the imported Simulink blocks as the implementation of these components. To clarify the use of this functionality, we import a Simulink design and select functional subsets which execute in discrete time, and then assign them to software components using a modeling language that has compatible (time-triggered) semantics. Communication links (signals) between Simulink blocks are mapped onto TTA messages passed between the tasks. The resulting language provides a componentized view of Simulink models that are scheduled periodically (with a fixed rate) and communicate using time-triggered messages. Extensions to heterogeneous MoC-s is an active area of research.

2.1 Requirements Analysis (RA)

Our example will model a data network implementing a single sensor/actuator loop with a distributed implementation. The sensors and actuators in the example are doubly-redundant, while the data network is triply-redundant. Unlike true safety-critical designs, we will deploy the same functions on all replicas rather than requiring multiple versions as is often done in practice [4]. The sensors and actuators close a single physical feedback loop. Specifying the physical system and particulars of the control functions are beyond the scope of this example as our focus is on modeling.

This example has an informal set of requirements, though our modeling language currently supports the formalization of timing constraints between sensor and actuator tasks. Formal requirements modeling offers great promise, but in ESMoL requirements modeling is still in conceptual stages. A simple sensor/actuator latency modeling example appears in a later section covering preliminary features for the language.

2.2 Functional Design (FD)

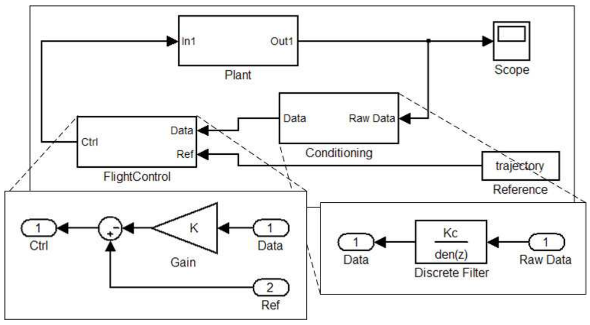


Fig. 2. Simulink design of a basic signal conditioner and controller.

Functional designs can appear in the form of Simulink/Stateflow models or as existing C code snippets. ESMoL does not support the full semantics of Simulink. In ESMoL the execution of Simulink data flow blocks is restricted to periodic discrete time, consistent with the underlying time-triggered platform. This also restricts the type and configuration of blocks that may be used in a design.

Continuous integrator blocks and sample time settings do not have meaning in ESMoL. C code snippets are captured in ESMoL as well. C code definitions are limited to synchronous, bounded-time function calls which will execute in a periodic task.

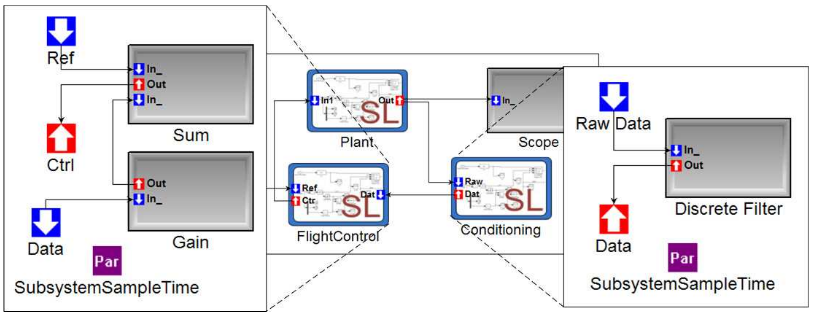


Fig. 3. ESMoL-imported functional models of the Simulink design.

Fig. 2 shows a simple top-level Simulink design for our feedback loop along with the imported ESMoL model (Fig. 3). The ESMoL model is a structural replica of the original Simulink, only endowed with a richer software design environment and tool-provided APIs for navigating and manipulating the model structure in code. A model import utility provides the illustrated function.

2.3 Software Architecture (SwA)

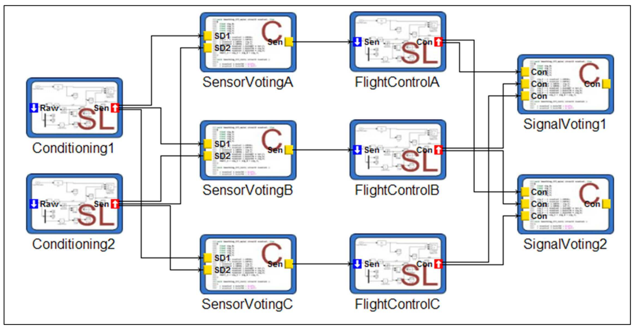


Fig. 4. The architecture diagram defines logical interconnections, and gives finer control over instantiation of functional units.

The software architecture model describes the logical interconnection of functional blocks. In the architecture language a component may be implemented by either a Simulink Subsystem or a C function. They are compatible at this level, because here their model elements represent the code that will finally implement the functions. These units are modeled as blocks with ports, where the ports represent parameters passed into and out of C function calls. The semantics for architecture model connections is that of sending and receiving messages using time-triggered communication.

Fig. 4 shows the architecture diagram for our TMR model. Instances of the functional blocks from the Simulink model are augmented with C code implementing replicated data voting.

2.4 Hardware Architecture (HwA)

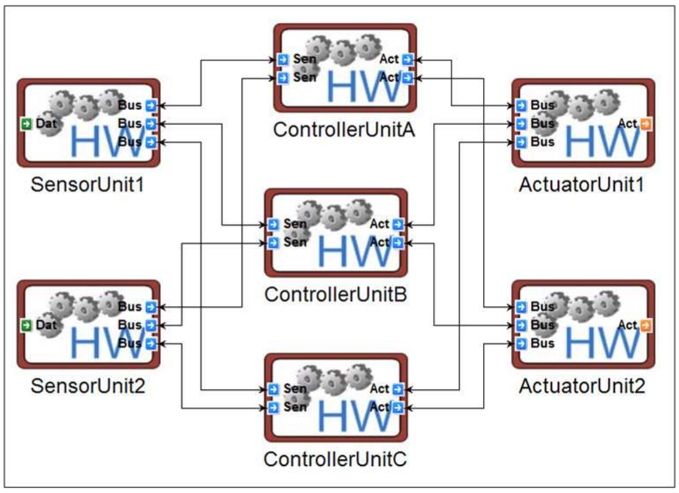


Fig. 5. Overall hardware layout for the TMR example.

Hardware configurations are explicitly modeled in the platform language. Platforms are defined hierarchically as hardware units with ports for interconnections. Primitive components include processing nodes and communication buses. Behavioral semantics for these networks come from the underlying time-triggered architecture. The platform provides services such as deterministic execution of replicated components and timed message-passing. Model attributes for hardware also capture timing resolution, overhead parameters for data transfers, and task context switching times.

Figs. 5 and 6 show model details for redundant hardware elements. Each controller unit is a private network with two nodes and three independent data buses. Sensor voting and flight control function instances will be deployed to the controller unit networks.

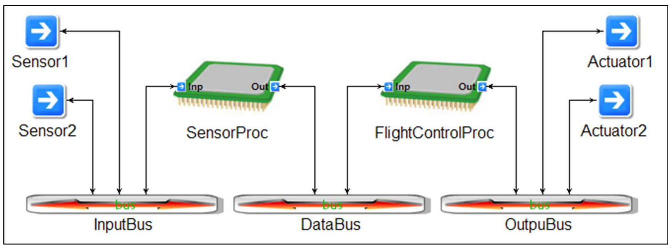


Fig. 6. Detail of hardware model for controller units.

2.5 Deployment Models (CD, SY, DPL)

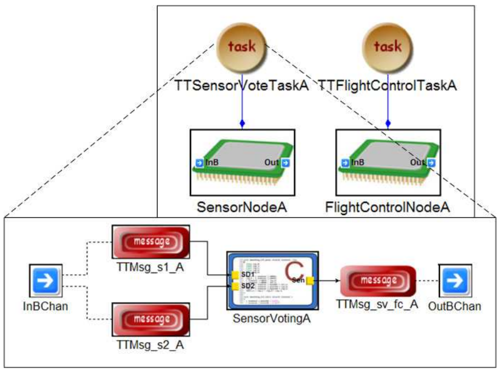


Fig. 7. Deployment model: task assignment to nodes and details of task definition.

A common graphical language captures the grouping of architecture components into tasks. In ESMoL a task executes on a single processing node at a single periodic rate. All components within the task execute synchronously. Data sent between tasks takes the form of messages in the model. Whether delivered locally (same processing node) or remotely, all inter-task messages are scheduled for delivery. ESMoL uses logical execution time semantics found in time-triggered languages such as Giotto [5] – message delivery is scheduled after the deadline of the sending task, but before the release of the receiving tasks. In the TT model of computation receivers assume that their data is available at task release time. Tasks never block, but execute with whatever data is available each period.

Deployment concepts, tasks running on processing nodes and messages sent over data buses, are modeled as shown in Fig. 7. Most of the model elements shown here are actually references to elements defined in the architecture and platform models. Model interpreters generate platform-specific code and analysis artifacts directly from the deployment models.

3 Existing Tools: Simulink to TTA

Control designs in Simulink are integrated using a graphical modeling language describing software architecture. Components within the architecture are assigned to tasks, which run on nodes in the platform.

3.1 Integration Details

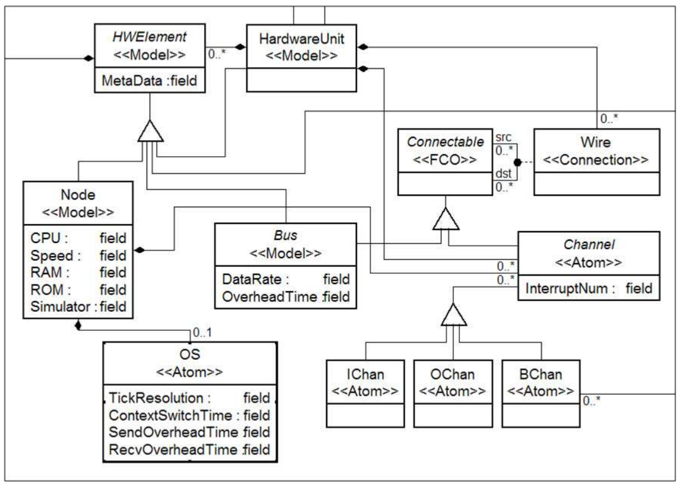


Fig. 8. Platforms. This metamodel describes a simple language for modeling the topology of a time-triggered processing network. A sample platform model is included.

The Simulink and Stateflow sublanguages of our modeling environment are described elsewhere, though the ESMoL language changes many of the other design concepts from older languages described by Neema [6].

In our toolchain we created a number of code generators. In the construction of the two main platform-independent code generators (one for Simulink-style models and another one for Stateflow-style models), we have used a higher-level approach based on graph transformations [7]. This approach relies on an assumption that (1) models are typed and attributed graphs with specific structure (governed by the metamodel of the language) and (2) executable code can be produced as an abstract syntax graph (which is then printed directly into source code). This graph transformation-based approach allows a higher-level representation of the translation process, which lends itself to algorithmic analysis of the transformations.

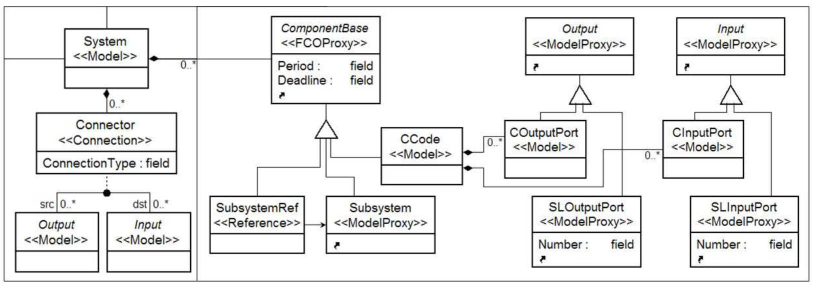


Fig. 9. Architecture Metamodel. Architecture models use Simulink subsystems or C code functions as components, adding attributes for real-time execution. The Input and Output port classes are typed according to the implementation class to which they belong.

The models in the example, and the metamodels described in the sequel were created using the ISIS Generic Modeling Environment tool (GME) [8]. GME allows language designers to create stereotyped UML-style class diagrams defining metamodels. The metamodels are instantiated into a graphical language, and metamodel class stereotypes and attributes determine how the elements are presented and used by modelers. The GME metamodeling syntax may not be entirely familiar to the reader, but it is well-documented elsewhere [9]. Class concepts such as inheritance can be read analogously to UML. Class aggregation represents containment in the modeling environment, though an aggregate element can be flagged as a port object. In the modeling environment a port object will also be visible at the next higher level in the model hierarchy, and available for connections. The dot between the Connectable class and the Wire class represents a line-style connector in the modeling environment.

High-confidence systems require platforms that provide services and guarantees for needed properties, e.g. fault containment, temporal firewalls, etc. These critical services (like partitioning) should be provided by the platform and not re-implemented from scratch by system developers [10]. Note that the platform also defines a 'Model of Computation' [11]. An MoC governs how the concurrent objects of an application interact (i.e. synchronization and communication), and how these activities unfold in time. The simple platform definition language shown in Fig. 8 contains relationships and attributes for describing a time-triggered network.

Similarly, Fig. 9 describes the software architecture language. The Connector element models communication between components. Semantic details of communication interactions remain abstract in this logical architecture – the platform model must be specified and associated in order to completely specify the interactions (though in this version we only offer synchronous and time-triggered communications).

Deployment models capture the assignment of Components (and Ports) from the Architecture to Platform Nodes (and Channels). Additional implementation

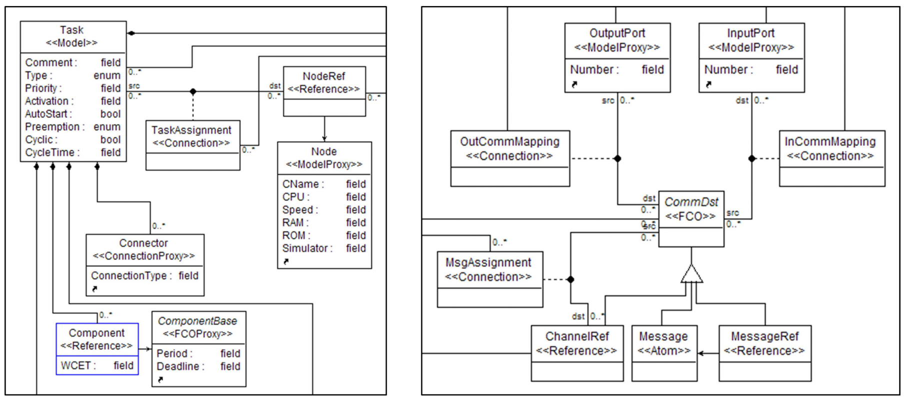


Fig. 10. Details from deployment sublanguage.

details (e.g. worst-case execution time) are represented here for platform-specific synthesis. Fig. 10 shows the relevant modeling concepts. Simulink objects SLInputPort and SLOutputPort are assigned to Message objects, which represent the marshaling of data to be sent on a Bus.

4 Under Development: Platform-specific simulation, generic hardware, and scheduling

A control system designer initially uses simulation to check correctness of the design. Software engineers later take code implementing control functions and deploy it to distributed controllers. Concurrent execution and platform limitations may introduce new behaviors which degrade controller performance and introduce errors. Ideally, the tools could allow the control functions to be re-simulated with appropriate platform effects.

The TrueTime simulation environment [12] provides Simulink blocks modeling processing nodes and communication links. Tasks can execute existing C code or invoke subsystems in Simulink models. Task execution follows configured real-time scheduling models, with communication over a selected medium and protocol. TrueTime models use a Matlab script to associate platform elements with function implementations. A platform-specific re-simulation requires this Matlab mapping function, and in our case also a periodic schedule for distributed time-triggered execution. Both of these can be obtained by synthesis from ESMoL models.

After resimulation follows synthesis to a time-triggered platform. In order to use generic computing hardware with this modeling environment, we created a simple, portable time-triggered virtual machine to simulate the timed behavior of a TT cluster [13] on generic processing hardware. Since the commercial TT cluster and the open TT virtual machine both implement the same model of computation, synthesis differences amount to management of structural details in the models. The open VM platform is limited to the timing precision of the underlying processor, operating system, and network, but it is useful for testing.

For both steps above the missing link is schedule generation. In commercial TTP platforms, associated software tools perform cluster analysis and schedule generation. For resimulation and deployment to an open platform, an open schedule generation tool is required. To this end we created a simple schedule generator using the Gecode constraint programming library [14]. The scheduling approach implements and extends the work of Schild and Würtz [15]. Configuration for the schedule generator is also generated by the modeling tools.

4.1 Integration Details

To configure TrueTime or the scheduler, the important details lie in the deployment model. Tasks and Messages must be associated with the proper processing nodes and bus channels in the model. The ISIS UDM libraries [16] provide a portable C++ API for creating interpreters to navigate models and extract the relevant information. See Fig. 10 for the relevant associations. Model navigation in these intepreters must maintain the relationships between processors and tasks and between buses and messages. Scheduler configuration also requires extraction of all message sender and receiver dependencies in the model.

5 Designs in Progress: Requirements and model updates

Many types of requirements apply to real-time embedded control systems design. Embedded systems are heterogeneous, so requirements can include constraints on control performance, computational resources, mechanical design, and reliability, to name a few things. Formal safety standards (e.g. DO-178B [4]) impose constraints on the designs as well as on the development process itself. Accordingly, current research has produced many techniques for formalizing requirements (e.g. ground models in abstract state machines [17] or Z notation [18]). Models could be used to incorporate formal requirements into other aspects of the design process. During analysis, requirements may appear as constraints in synthesized optimization problems or conditions for model checking. Requirements can also be used for test generation and assessment of results.

Management of model updates is also essential. As designs evolve engineers and developers reassess and make modifications. Changes to either the platform model or functional aspects of the design may invalidate architecture and deployment models created earlier. Some portions of the dependent models will survive changes. Other parts needing changes must be identified. Where possible, updates should be automated.

5.1 Integration Details

The requirements sublanguage is in design, and so is light on details. Fig. 13 shows an example model with latency requirements between tasks, and Fig. 11

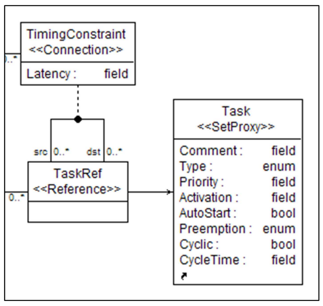


Fig. 11. Latencies are timing constraints between task execution times.

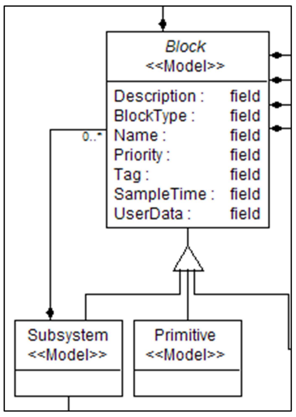


Fig. 12. Simulink's UserData field can help manage model changes occuring outside the design environment.

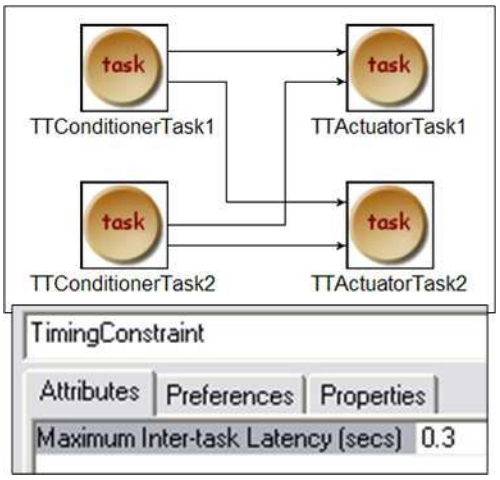
shows the modeling language definition. This simple relationship can be quantified and passed directly to the schedule solver as a constraint. Ideally a more sophisticated requirements language could capture the syntax and semantics of an existing formal requirements tool. Some candidate languages and approaches are currently under consideration for inclusion in the framework.

To track model changes we propose to use the Simulink UserData field to store unique tags when the models are imported. During an update operation tags in the control design can be compared with previously imported tags in the model environment. Fig. 12 shows the UserData attribute from our Simulink sublanguage, corresponding to the actual attribute in Simulink blocks. To handle issues arising from topology concerns, we require control designers to group top-level functionality into subsystems and place a few restrictions on model hierarchy in deployment models.

6 Wishlist: Expanded semantics, implementation generation, and verification

Many exciting possibilities loom on the horizon for this tool chain construction effort. We briefly describe some forward-looking concepts currently in discussion for the tools.

The current modeling languages describe systems which provide performance and reliability guarantees by implementing a time-triggered model of computation. This is not adequate for many physical processes and controller platforms. We also need provisions for event-triggered communication and components. Event-triggered component structures give rise to interesting and useful communication patterns common in practical systems (e.g. publish-subscribe, ren-



Maximum Inter-task Latency (secs) 0.3

Fig. 13. Example of task latency spec for sample model, with detail of timing attribute value specified on model links.

dezvous, and broadcast). Several research projects have explored heterogeneous timed models of computation. Two notable examples are the Ptolemy project [19] and the DEVs formalism and associated implementations [20]. More general simulation and model-checking tools for timed systems and specifications include UPPAAL [21] and timed abstract state machines [22]. We aim to identify useful design idioms from event-triggered models and extend the semantics of the modeling language to incorporate them. Synthesis to analysis tools is also possible using model APIs.

Safe automation of controller implementation techniques is another focus. Control designs are often created and simulated in continuous time and arbitrary numerical precision, and then discretized in time for platforms with periodic sampling and in value for platforms with limited numeric precision. Recent work in optimization and control offers some techniques for building optimization problems which describe valid controller implementation possibilities [23] [24]. Early model interpreter work aims to generate such optimization problems directly from the models. Other interesting problems include automated generation of fixed-point scaling for data flow designs. If integrated, tools like BIP [25] provide potential for automated verification of distributed computing properties (safety, liveness, etc...). Model representation of data flow functions, platform precision, and safety requirements could be used together for scaling calculation.

The addition of proper formal requirements modeling can enable synthesis of conditions for model checking and other verification tools. Executable semantics for these modeling languages can also provide the behavioral models to be checked (see Chen [26] [27], Gargantini [28], and Ouimet [29]). Other relevant work includes integration of code-level checking, as in the Java Pathfinder [30] or Saturn [31] tools. Synthesis to these models must also be verified, an active area of research at ISIS [32].

7 Acknowledgements

This work is sponsored in part by the National Science Foundation (grant NSF-CCF-0820088) and by the Air Force Office of Scientific Research, USAF (grant/contract number FA9550-06-0312). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

1. Henzinger, T., Sifakis, J.: The embedded systems design challenge. In: FM: Formal Methods. Lecture Notes in Computer Science 4085. Springer (2006) 1–15
2. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software (Oct 2001)
3. AS-2 Embedded Computing Systems Committee: Architecture analysis and design language (aadl). Technical Report AS5506, Society of Automotive Engineers (November 2004)
4. RTCA, Inc. 1828 L St. NW, Ste. 805, Washington, D.C. 20036: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. (December 1992) Prepared by: RTCA SC-167.
5. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Lecture Notes in Computer Science **2211** (2001) 166–184
6. Neema, S., Karsai, G.: Embedded control systems language for distributed processing (ECSL-DP). Technical Report ISIS-04-505, Institute for Software Integrated Systems, Vanderbilt University (2004)
7. Aditya Agrawal and Gabor Karsai and Sandeep Neema and Feng Shi and Attila Vizhanyo: The design of a language for model transformations. Journal on Software and System Modeling **5**(3) (Sep 2006) 261–288
8. ISIS, V.U.: Generic Modeling Environment. http://repo.isis.vanderbilt.edu/
9. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (Jan. 2003)
10. Sangiovanni-Vincentelli, A.: Defining Platform-based Design. EEDesign of EE-Times (February 2002)
11. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A denotational framework for comparing models of computation. Technical Report UCB/ERL M97/11, EECS Department, University of California, Berkeley (1997)
12. Ohlin, M., Henriksson, D., Cervin, A.: TrueTime 1.5 Reference Manual. Dept. of Automatic Control, Lund University, Sweden. (January 2007) http://www.control.lth.se/truetime/.
13. Thibodeaux, R.: The specification and implementation of a model of computation. Master's thesis, Vanderbilt University (May 2008)
14. Schulte, C., Lagerkvist, M., Tack, G.: Gecode: Generic Constraint Development Environment. http://www.gecode.org/

15. Schild, K., Würtz, J.: Scheduling of time-triggered real-time systems. Constraints **5**(4) (Oct. 2000) 335–357
16. Magyari, E., Bakay, A., Lang, A., et al: Udm: An infrastructure for implementing domain-specific modeling languages. In: The 3rd OOPSLA Workshop on Domain-Specific Modeling. (October 2003)
17. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
18. ISO/IEC: Information Technology Z Formal Specification Notation Syntax, Type System and Semantics. (July 2002) 13568:2002.
19. UCB: Ptolemy II. `http://ptolemy.berkeley.edu/ptolemyII/`
20. Hwang, M.H.: DEVS++: C++ Open Source Library of DEVS Formalism, http://odevspp.sourceforge.net/. first edn. (May 2007)
21. Basic Research in Computer Science (Aalborg Univ.) / Dept. of Information Technology (Uppsala Univ.): Uppaal. http://www.uppaal.com/ Integrated tool environment for modeling, validation and verification of real-time systems.
22. Ouimet, M., Lundqvist, K.: The timed abstract state machine language: An executable specification language for reactive real-time systems. In: Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07), Nancy, France (March 2007)
23. Skaf, J., Boyd, S.: Controller coefficient truncation using lyapunov performance certificate. IEEE Transactions on Automatic Control (in review) (December 2006)
24. Bhave, A., Krogh, B.H.: Performance bounds on state-feedback controllers with network delay. In: IEEE Conference on Decision and Control, 2008 (submitted). (December 2008)
25. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, IEEE Computer Society (2006) 3–12
26. Chen, K., Sztipanovits, J., Abdelwahed, S.: A semantic unit for timed automata based modeling languages. In: Proceedings of RTAS'06. (2006) 347–360
27. Chen, K., Sztipanovits, J., Abdelwahed, S., Jackson, E.: Semantic anchoring with model transformations. In: Proceedings of European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA). Volume 3748 of Lecture Notes in Computer Science., Nuremberg, Germany, Springer-Verlag (November 2005) 115–129
28. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from asm specifications. In: Abstract State Machines 2003: Advances in Theory and Practice, 10th International Workshop. Volume 2589 of Lecture Notes in Computer Science., Springer (March 2003) 263–277
29. Ouimet, M., Lundqvist, K.: Automated verification of completeness and consistency of abstract state machine specifications using a sat solver. In: 3rd International Workshop on Model-Based Testing (MBT '07), Satellite of ETAPS '07, Braga, Portugal (April 2007)
30. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering Journal **10**(2) (April 2003)
31. Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug detection. In: Proceedings of the 17th International Conference on Computer Aided Verification. (January 2005) 139–143
32. A. Narayanan and G. Karsai: Towards verifying model transformations. In R. Bruni and D. Varr, ed.: 5th International Workshop on Graph Transformation and Visual Modeling Techniques, 2006, Vienna, Austria. (Apr 2006) 185–194

Modeling Radio-Frequency Front-Ends Using SysML: A Case Study of a UMTS Transceiver

Sabeur Lafi, Roger Champagne, Ammar B. Kouki, and Jean Belzile

École de technologie supérieure, 1100 rue Notre-Dame Ouest, Montréal, H3C 1K3, Canada.

Abstract. Numerous engineering fields are nowadays dealing with complex systems. The analysis, design, testing and maintenance of such systems are crucial challenges. For this purpose, the OMG proposed SysML, an extension of UML, in order to address the issues of modeling complex systems in different engineering domains. This standard enables the elaboration of efficient tools allowing automated analysis, verification and validation of systems. The radio-frequency front-end's design is one of engineering fields which would benefit from such a technology to enhance the efficiency of the design and manufacturing process. In this paper, we discuss the provision and the limitations of both UML and SysML. We also present a case study consisting of the modeling of a Universal Mobile Telecommunications System (UMTS) transceiver using SysML and we discuss the advantages and the drawbacks of such a technology from the designer's point of view.

Keywords: SysML, UML, Modeling, Systems Engineering, UMTS, Transceiver.

1 Introduction

Engineering systems are increasingly growing in complexity, implying various design and testing challenges. Consequently, multiple fields of engineering are looking for a general-purpose and high-level methodology for systems' modeling. This can effectively enable an efficient design process from specifications all the way through to delivery and maintenance. One of these fields is radio-frequency (RF) and microwave engineering which mainly addresses the design and manufacturing of microwave radios and components. In fact, RF front-ends represent an important part in several embedded devices such as wireless sensors and smart radios.

Modern RF front-ends need to be modeled in parallel with the baseband hardware and software parts which carry out signal processing and support, in some cases, user applications. Software modeling can currently be achieved using Unified Modeling Language (UML) [1]. UML was originally defined by the Object Management Group (OMG) in order to enable the definition and modeling of complex software systems and was later used in other fields. On the other hand, complex engineering systems including software, electrical, hydraulic and mechanical hardware, can be modeled by the recent Systems Modeling Language (SysML) [2].

Because SysML is a recent standard addressing modeling in a wide range of domains, each engineering field must first evaluate its abilities to express and describe its specific particularities. Some studies had been already carried out in some fields such as sensor networks [3] and System-on-Chip/Network-on-Chip [4]. As far as RF design is concerned, case studies must be performed to test the usefulness of SysML for modeling RF systems. In this paper, we first focus on the modeling languages and we compare the use of OMG's UML and SysML in the modeling of software/hardware systems. We specifically discuss how RF front-ends can be modeled using SysML. Second, we present a case study in which a UMTS transceiver is modeled using SysML. Finally, we discuss the benefits and the limitations of using SysML in RF systems' design.

2 Modeling Languages in Modern Systems Engineering

The growing complexity of software systems led various organizations and research task groups, both in industry and academia, to investigate modeling techniques. One of these organizations, the OMG, has gathered several proposals with the intent of elaborating a standardized modeling language [5], [6], [7]. The outcome of this effort was the establishment of the Unified Modeling Language (UML). Therefore, UML, a visual specification language for object modeling, has emerged as a viable modeling language empowering software design, bringing high-level of abstraction and enabling different automated techniques such as code generation, verification and validation of software and allowed also data interchange and meta-modeling.

UML has proven its importance particularly in the field of software design. Its extensibility enhanced its scope to other domains. This advantage gives more appeal to UML which can then be used to model systems other than software. However, the provision of UML to systems engineering is limited. In fact, it is unable to express specific aspects of several domains. For example, it does not support efficiently the modeling of dynamically changing parameters causing the system to behave differently under different configurations [3]. It also expresses weakly the relationships between mixed systems composed of both hardware and software [3]. An interesting survey about the common limitations and defects of UML is given in [8]. In order to address these limitations and others, the OMG has specified and standardized another modeling language, SysML, which is intended to provide visual modeling support for a wide range of engineered systems. SysML is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include different types of components such as software, hardware, etc. [2]. It provides graphical representations with semantic foundations in order to model different aspects of a complex system such as its structure, behavior, requirements, and parametrics.

Despite the fact that SysML is a subset of UML, it differs remarkably from it. First, UML is a general-purpose modeling language while SysML, as a customized profile of UML 2.0 conceived for systems engineering, is domain-specific. Thus, SysML is smaller and easier to learn than UML since it removes many software-centric constructs. It expresses systems engineering semantics better than UML. Second, it is a precise language, including support for constraints and parametric analysis which allows models to be analyzed and simulated, greatly improving the value of system models compared to textual system descriptions. SysML also supports various diagrams that facilitate automated analyses, verification and validation. In addition, it is an open standard which is compliant with various data interchange formats such as XML, XMI (XML Metadata Interchange) and AP233 standards. Furthermore, SysML improves communication by using a formal language, namely Object Constraint Language (OCL), for sharing system information to all project engineers [9], [10].

RF front-ends, as a complex engineering domain, require high-level modeling methodologies providing enough flexibility and abstraction in order to analyze, design, and validate RF systems. In practice, modeling would help the automation of several design aspects. Some of the most important are (i) verification, as the process of determining that a model implementation accurately represents the designer's conceptual description of the model and (ii) validation, as the process of determining the degree to which a model is an accurate representation of the real RF system from a functional perspective. In this context, we try to explore the provision and the limitations of SysML in this regard. For this purpose, we present in the next sections the results of a case study consisting of the modeling of a UMTS transceiver using SysML.

3 Case Study: Modeling a UMTS Transceiver Using SysML

To evaluate the benefits and the limitations of SysML in modeling RF and microwave front-ends, which represent an important interface between embedded systems and the real world, we propose to apply it to the modeling of a UMTS transceiver. This choice is motivated by the fact that a UMTS mobile phone is composed of three main parts: (i) software, carrying out the main signal processing, internetworking and user applications, (ii) digital hardware, such as digital signal processors to execute the signal processing algorithms and baseband operations and (iii) analog hardware, carrying out the transmission/reception of radio signals to/from the base station, known as node B in the UMTS terminology. If software can be modeled using UML and digital hardware using hardware description languages, analog hardware is still developed using classical techniques. As a result, RF design lacks flexibility and is still implemented manually. The prospect of modeling it using SysML is to achieve a high-level of abstraction and automation, particularly for such RF design tasks as verification and validation. In this section, we present a summary of the UMTS transceiver specifications and describe how we capture and model them using SysML. Due to space constraints, only a subset of the SysML diagrams we experimented with for this system is presented.

3.1 UMTS Transceiver Specifications

Universal Mobile Telecommunication System (UMTS) is a mobile standard conceived for third-generation mobile communications networks. This communication standard had specified different radio interfaces. In this section, we present a summary of the specification for a UMTS Terrestrial Radio

Access/Frequency Division Duplex (UTRA/FDD) compliant mobile transceiver. This summary is based on [11], [12] and [13].

An UTRA/FDD transceiver is a radio whose RF front-end is composed of three parts: (i) duplex filter, (ii) transmitter and (iii) receiver. All three are linked to the antenna as shown in Fig. 1.

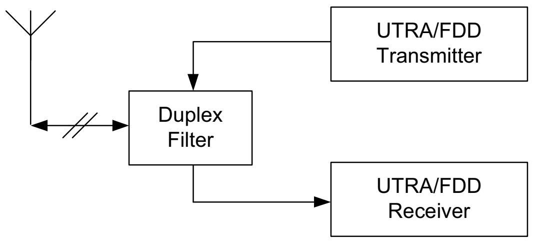


Fig. 1. A synoptic schematic of an UTRA/FDD Transceiver.

There are various architectures for implementing the transmitter and receiver blocks. Among these, direct up-/down-conversion architectures are being widely used in mobile communications, particularly in GSM and W-CDMA applications [11]. In fact, a direct up-/down-conversion transceiver is less complicated than classical architectures such as a superheterodyne radio. Fig. 2 shows a typical architecture of a direct conversion UTRA/FDD transceiver.

In the transmission phase, the baseband signal is modulated and data symbols are typically converted into two analog signals (in-phase, *I*, and in quadrature, *Q*) via digital-to-analog converters. The *I*/*Q* signals are then up-converted in the quadrature up-conversion stage (QUC) to the RF frequency determined by the local oscillator (LO) and then summed. The resulting signal is then amplified, by the variable-gain amplifier (VGA), in order to adjust its power level to the desired value, and filtered by the band-pass filter (BPF-T), in order to reduce intermodulation products. The final stage of the transmitter is the power amplifier (PA) which amplifies the transmitted signal to the high power level required for transmission. In the reception phase, the received signal is low power and its level is close to the noise floor. Therefore, it is first amplified by the low-noise amplifier (LNA1), which keeps the added noise to a minimum, and pass-band filtered by BPF-R in order to eliminate interferences. A second stage amplifier, LNA2, boosts the received signal further before it is divided into two signals which are mixed with the LO reference signal. One signal is mixed with the inphase LO while the other with the quadrature (90-degree phase-shifted) LO. The resulting *I*/*Q* signals are then amplified and filtered, to remove intermodulation products and non desirable signals, in the BFA1 and BFA2 blocks before they are digitized for baseband demodulation.

In a UTRA/FDD transceiver, both the receiver and the transmitter operate simultaneously. Any leakage between the transmitter and the receiver can either saturate the low-noise amplifier or disturb the transmitted signal. For this purpose, a duplex filter is added in order to isolate the received and the transmitted signals. The UTRA/FDD standard determines rigorously the specifications of the duplex filter and each component in the receiver/transmitter chains. For example, Table 1 presents the specifications of the duplex filter [11] while Table 2 presents some key specifications of a UTRA/FDD radio as stated in [13]. The duplex filter plays the role of a duplexer and a filter. It isolates the incoming and outgoing signals and also allows the rejection of out-of-band interferences. This duality in role implies severe constraints in terms of isolation and in-band attenuation.

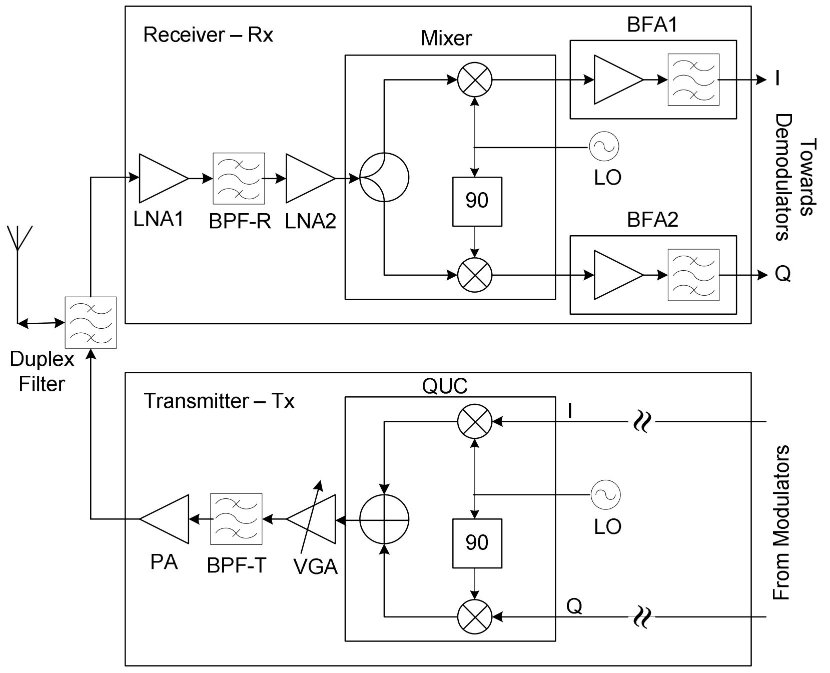


Fig. 2. A detailed view of a direct-conversion UTRA/FDD transceiver.

Table 1. Duplex filter requirements in terms of attenuation and isolation.

Tx – Antenna Attenuation (dB)	< 2
Rx – Antenna Attenuation (dB)	< 3
Tx – Rx Isolation / Tx-Band (dB)	> 50
Tx – Rx Isolation / Rx-Band (dB)	> 37

Table 2. Some of UMTS standard specifications.

Frequency Band (Up-link) (MHz)	1920 – 1980
Frequency Band (Down-link) (MHz)	2110 – 2170
Frequency Spacing (MHz)	4.4 – 5.2
Modulation	QPSK
Pulse Shaping	RRC / roll-off = 0.22
Chip Rate (Mc/s)	3.84
User Bit Rate (kbps) @ BER=10^{-3}	12.2
Power Control Frequency (Hz)	1500

Signal-to-Noise Ratio (dB)	6.0
Power Sensitivity (dBm)	-120.0
Input Noise Level (dBm)	-111.0
Reference LO Power (dBm)	6.0

The UMTS standard specifies the baseband characteristics such as the type of modulation, shaping filter and the chip rate. It also establishes the front-end parameters such as the input noise level, the signal-to-noise ratio, the power sensitivity, etc. These specifications are generally produced following extensive system level analysis. In the next section, we present how to capture the UTRA/FDD specifications using a SysML model.

3.2 UTRA/FDD Transceiver's SysML Model

We presented some of the specifications of a UTRA/FDD transceiver in the previous section. We chose the direct conversion architecture to implement the radio. For the detailed specifications, the reader can refer to [11], [12] and [13]. In this section, we present how to capture these specifications in a SysML model. In this model, we present the structure of the overall transceiver and we detail the internal blocks of the receiver. We also show how to capture some of the transceiver requirements.

The structure of a RF front-end incorporates the different components of which it is made. The SysML model can capture this structure using different diagrams at different levels of abstraction. Among these diagrams, we use the *package diagram* in order to give an overview of the general structure of the model packages, see Fig. 3. As shown in this Figure, the transceiver's SysML model is organized into four main packages: (i) Value Types, describing the measurement units used in the other packages (ii) Transceiver Structure, describing the structural components of the transceiver (iii) Transceiver Behavior, describing the signal flow inside the transceiver (iv) Transceiver Requirements, illustrating the requirements of the transceiver.

As in many disciplines, different measurement units are used in the radio-frequency domain. SysML allows their modeling using "*value types*". In the value types package, shown in detail in Fig. 4, we present the measurement units used in the specifications of a UTRA/FDD transceiver. All other packages using these measurement units have a "*dependency*" relationship with this package (see Fig. 3).

The structure package is composed of diagrams such as the *block definition diagram* and the *internal block diagram*. The former illustrates the structure of an object with blocks presenting its different components while the latter gives an insight of how a block is structured. The general structure of the UTRA/FDD transceiver can be modeled using a standard block definition diagram. This diagram captures the different components of the transceiver and organizes them into different levels of hierarchy. Fig. 5 shows the block definition diagram of the entire transceiver which is made up of a back-end consisting in an antenna, and a RF front-end which includes the duplex filter, the transmitter and the receiver. The duplex filter is an atomic component. However, the receiver and the transmitter are composed of several other components such as the local oscillator, the mixer, etc. To illustrate the structure of the transceiver for example, we can use the internal block diagram as shown in Fig. 6.

The components of the blocks are called "*parts*". One of the advantages of this representation is its ability to capture how they are connected and which types of information or signals can be exchanged between them.

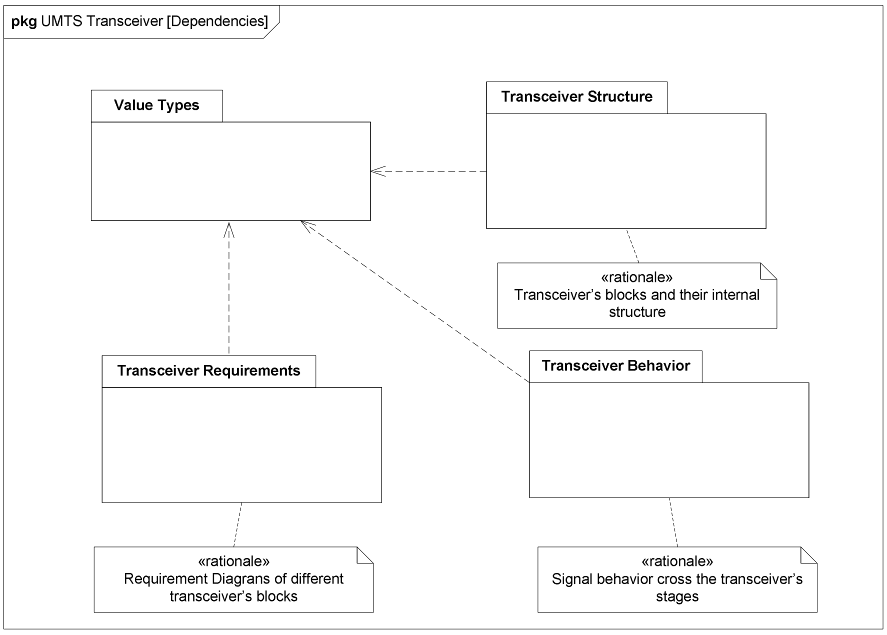

pkg UMTS Transceiver [Dependencies]
Value Types
Transceiver Structure
«rationale»
Transceiver's blocks and their internal structure
Transceiver Requirements
Transceiver Behavior
«rationale»
Requirement Diagrans of different transceiver's blocks
«rationale»
Signal behavior cross the transceiver's stages

Fig. 3. Package diagram of the UTRA/FDD transceiver's SysML model.

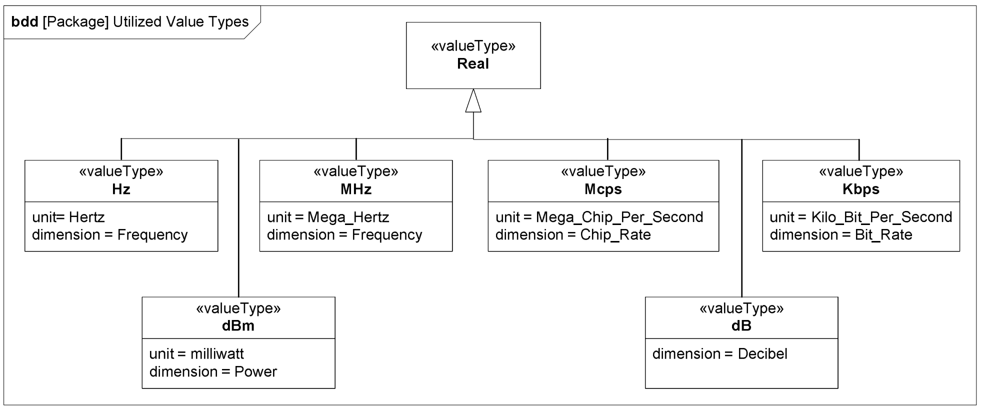


Fig. 4. Value types package.

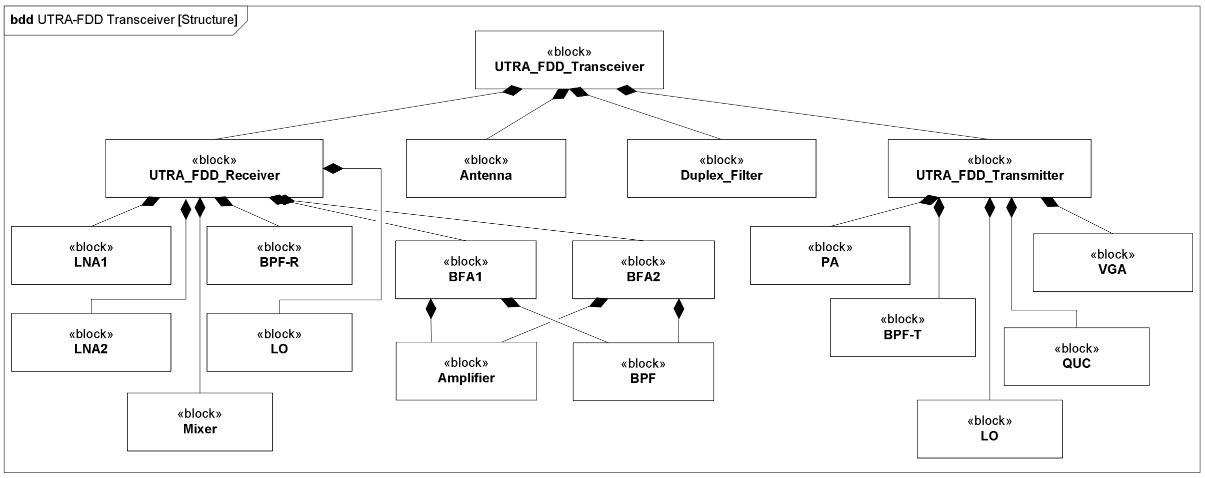


Fig. 5. The block definition diagram of the UTRA/FDD transceiver showing the hierarchy of its components.

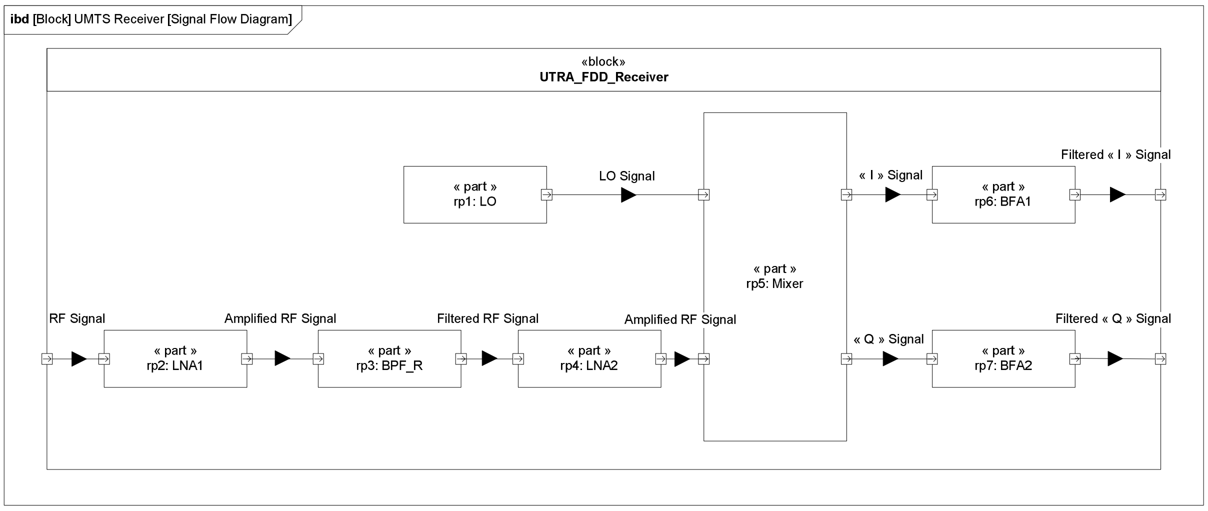


Fig. 6. The internal block diagram of the receiver presenting how its different parts are linked as well as the signal flow between them.

Different analog signals of different origins travel between these components. For example, two types of signals are required inside the mixer block. Both the baseband and the LO reference signals are needed in order to achieve the down-conversion operation. These signals are communicated to the mixer via its input ports. The flow of signals can also be captured by the internal block diagram as shown in Fig. 7. The RF signal is communicated to the mixer by its RF input port and then divided in two signals: one is in-phase and another is 90-degree phase-shifted. Both of them are down-converted according to the LO reference frequency received from the LO output port. The result is the *I* and *Q* signals, each carrying a part of the information.

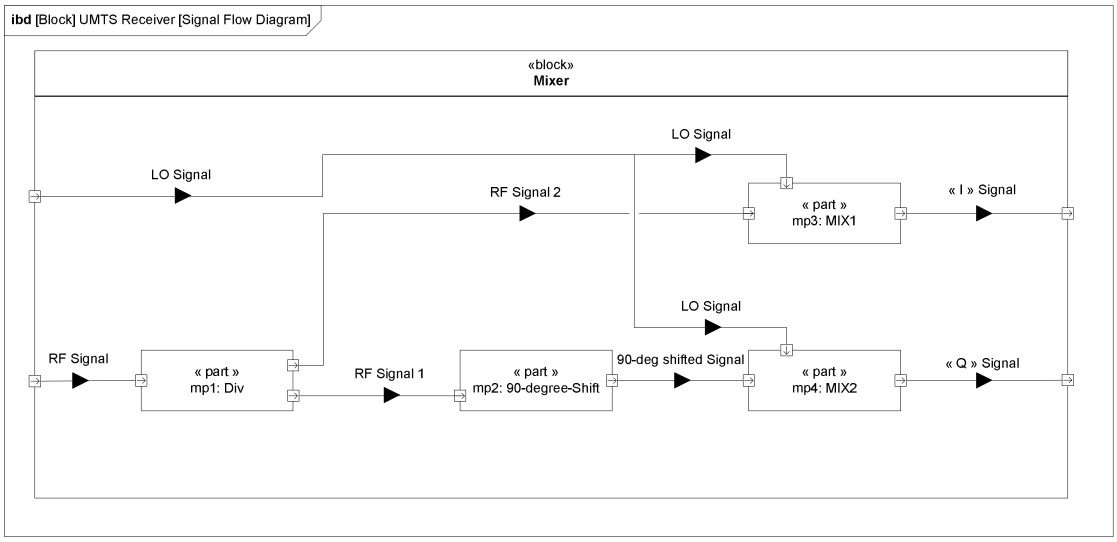


Fig. 7. The internal block diagram of the receiver's mixer and the signal flow inside it.

One of SysML novelties is the *requirement diagram* whose role is the capture of specifications and requirements of an engineering system in a simple and standard manner. In RF engineering, the requirement diagram is an important tool that can help the designer to represent and communicate the specifications to the other designers of the team. For example, the information of Table 1 can be represented in a requirement diagram as shown in Fig. 8. Requirements can be organized in a hierarchal fashion. They can be copied, derived or traced. Test cases can be added in order to verify the system at the end of the design cycle. In the example of Fig. 8, the duplex filter has two main properties: attenuation and isolation. Its final design must satisfy the requirements expressed in Table 1.

Requirements can be organized in a nested structure, as shown in Fig. 9. This allows more clarity in the representation. They can also be grouped into constraints, test cases, etc. For example, in Fig. 9 the user bit rate is considered 12.2 kbps only if the corresponding bit error rate (BER) is equal to 10^{-3}.

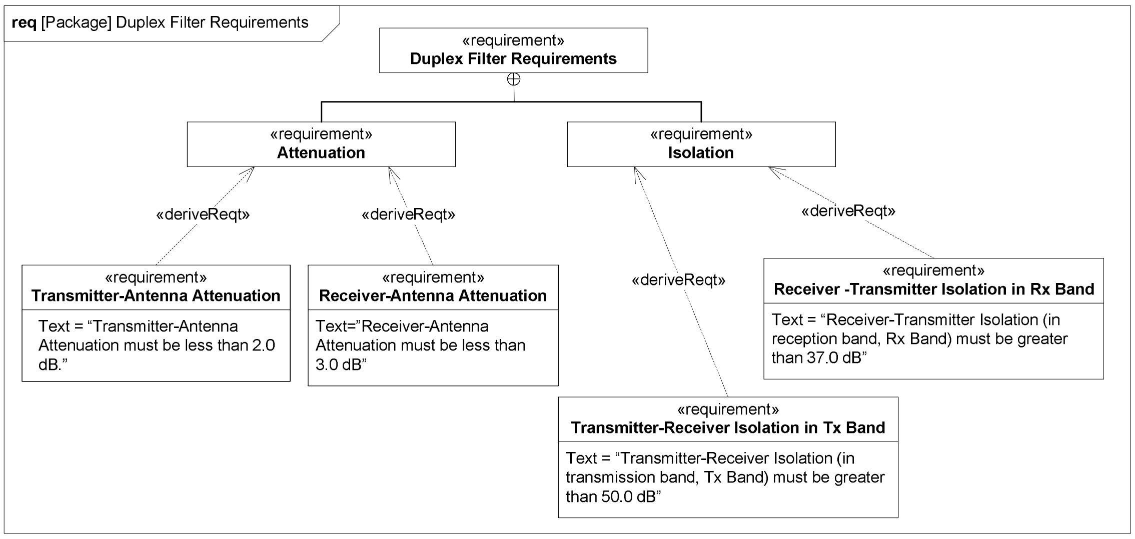

req [Package] Duplex Filter Requirements
«requirement» Duplex Filter Requirements
«requirement» Attenuation
«requirement» Isolation
«deriveReqt»
«deriveReqt»
«deriveReqt»
«deriveReqt»
«requirement» Transmitter-Antenna Attenuation
Text = "Transmitter-Antenna Attenuation must be less than 2.0 dB."
«requirement» Receiver-Antenna Attenuation
Text="Receiver-Antenna Attenuation must be less than 3.0 dB"
«requirement» Receiver -Transmitter Isolation in Rx Band
Text = "Receiver-Transmitter Isolation (in reception band, Rx Band) must be greater than 37.0 dB"
«requirement» Transmitter-Receiver Isolation in Tx Band
Text = "Transmitter-Receiver Isolation (in transmission band, Tx Band) must be greater than 50.0 dB"

Fig. 8. The requirement diagram of the duplex filter.

4 Discussion

In the previous section, we presented a model of a UTRA/FDD transceiver. We have shown particularly how the different aspects of a RF system's specifications can be captured using a SysML model. In this section, we discuss the consistency and coherency of such a methodology for modeling RF front-ends. We specifically focus on the provision and the limitations observed in this case study.

SysML is a language, inspired from UML, aiming to offer a powerful standard supporting rigorous modeling of various systems. It addresses this issue in a wide range of engineering domains. This mission is not easy because each engineering field has its own particularities. Despite the fact that SysML is defined as a new language, it retains and extends many concepts of UML. A legitimate question is: is this property a provision or a limitation?

In the radio-frequency domain, semantics are very important. The notations and the representations are needed by RF designers and engineers for easily expressing, understanding and sharing designs. For example, the symbols used to represent RF components such as mixers and LOs are fixed by a consensus. This facilitates the understanding and the interpretation of RF schematics. However, their SysML representation, being currently limited to a restrictive set of notations, lacks the flexibility of customized symbolic representation. Consequently, RF engineers will find it difficult to read and interpret SysML diagrams and impractical to work with its notation. Adding customizable symbols to the SysML standard representation of blocks and parts will go a long way towards making SysML more easily accepted by

RF engineers. To illustrate this, we reproduce the block definition diagram of Fig. 5 with added RF symbols as shown in Fig. 10. This enhances considerably the readability of this SysML representation of the transceiver. A possible solution to adopt in this regard is the creation of a SysML profile for modeling RF and analog components. Such a profile would define the stereotypes and constraints which enable the specifications, analysis and verification of RF systems.

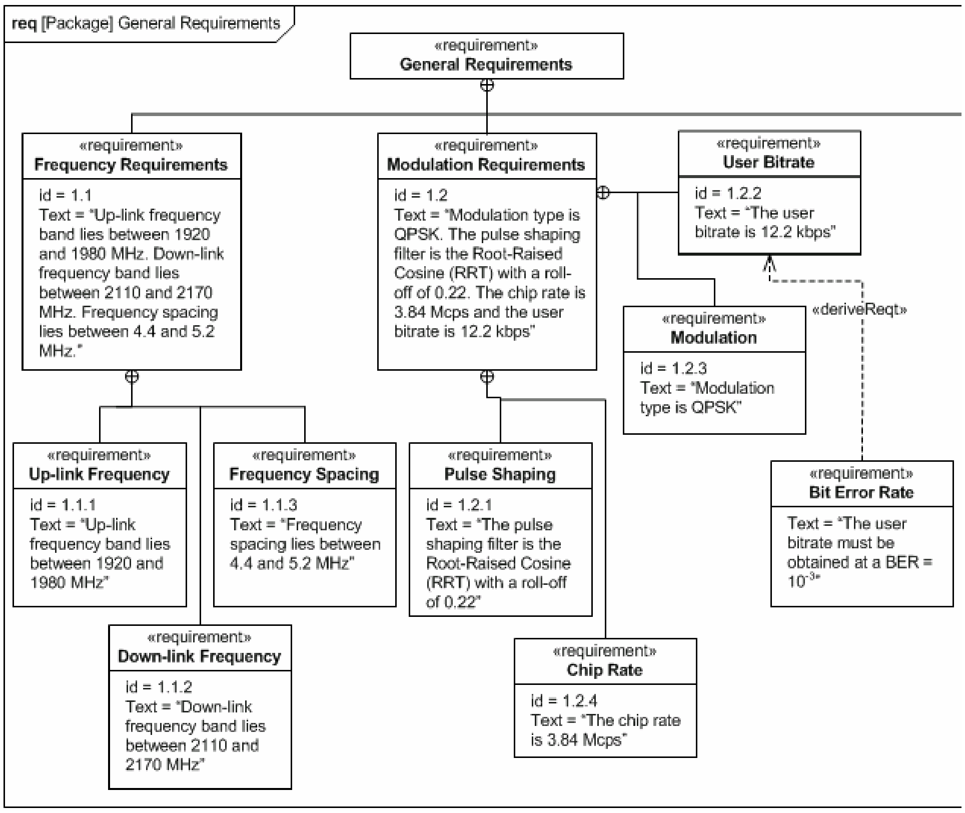

req [Package] General Requirements
«requirement» General Requirements
«requirement» Frequency Requirements
id = 1.1
Text = "Up-link frequency band lies between 1920 and 1980 MHz. Down-link frequency band lies between 2110 and 2170 MHz. Frequency spacing lies between 4.4 and 5.2 MHz."
«requirement» Modulation Requirements
id = 1.2
Text = "Modulation type is QPSK. The pulse shaping filter is the Root-Raised Cosine (RRT) with a roll-off of 0.22. The chip rate is 3.84 Mcps and the user bitrate is 12.2 kbps"
«requirement» User Bitrate
id = 1.2.2
Text = "The user bitrate is 12.2 kbps"
«deriveReqt»
«requirement» Modulation
id = 1.2.3
Text = "Modulation type is QPSK"
«requirement» Up-link Frequency
id = 1.1.1
Text = "Up-link frequency band lies between 1920 and 1980 MHz"
«requirement» Frequency Spacing
id = 1.1.3
Text = "Frequency spacing lies between 4.4 and 5.2 MHz"
«requirement» Pulse Shaping
id = 1.2.1
Text = "The pulse shaping filter is the Root-Raised Cosine (RRT) with a roll-off of 0.22"
«requirement» Bit Error Rate
Text = "The user bitrate must be obtained at a BER = 10^-3"
«requirement» Down-link Frequency
id = 1.1.2
Text = "Down-link frequency band lies between 2110 and 2170 MHz"
«requirement» Chip Rate
id = 1.2.4
Text = "The chip rate is 3.84 Mcps"

Fig. 9. A portion of the requirement diagram expressing the specifications of Table 2.

On the modeling level, some important questions remain open, namely, what is the right depth of a SysML model? In other words, is there any definition of the granularity concept? At this stage, it is difficult to formulate a definite answer since (i) SysML, like UML, is a notation and not a methodology, (ii) a SysML model is not unique and (iii) many experiments and case studies have to be carried out in order to learn how the depth of a model relates to the hierarchical levels of the modeled system. Consequently, a SysML model depends on the level of experience of the system modeler and may need several iterations before reaching acceptable results.

In the light of the above case study, one can argue that the provision of SysML to the design of RF systems lies in three main levels: (i) abstraction, (ii) flexibility and (iii) requirements. First, on the abstraction level, SysML can represent the structure of a RF system in different ways. Aspects such as hierarchy, containment, and multiplicity can be expressed rigorously. This allows the masking of some levels of the SysML model. Such a mechanism can be very useful in RF systems. In fact, one of the issues in RF systems is the absence of an abstraction mechanism which controls the level of details of the system meaning that the designer can choose the level of abstraction and the granularity at which he wants to carry out the analysis of the system. Such a mechanism can really empower a hardware abstraction strategy allowing automatic design and synthesis of RF components and systems. Second, our experience has shown that SysML formalism and notations are flexible enough to express most of a RF system's aspects. For example, the port is considered as the lowest level of abstraction in a RF system. SysML allows describing the properties of RF ports, the flows travelling between them and the connectors relating them. Third, SysML can capture and express requirements in an organized and simple way. For several years, designers have been experiencing difficulties communicating the specifications among themselves. Ambiguities and forgotten details usually lead to serious negative effects on the design, test, integration and validation times. SysML presents an important evolution from traditional requirements management tools to UML/SysML models which offer a rich language for expressing the context, behavior and constraints of an engineering system. Therefore, the requirement diagram of SysML can be a useful tool to help RF designers and engineers to organize their specifications in a rigorous way.

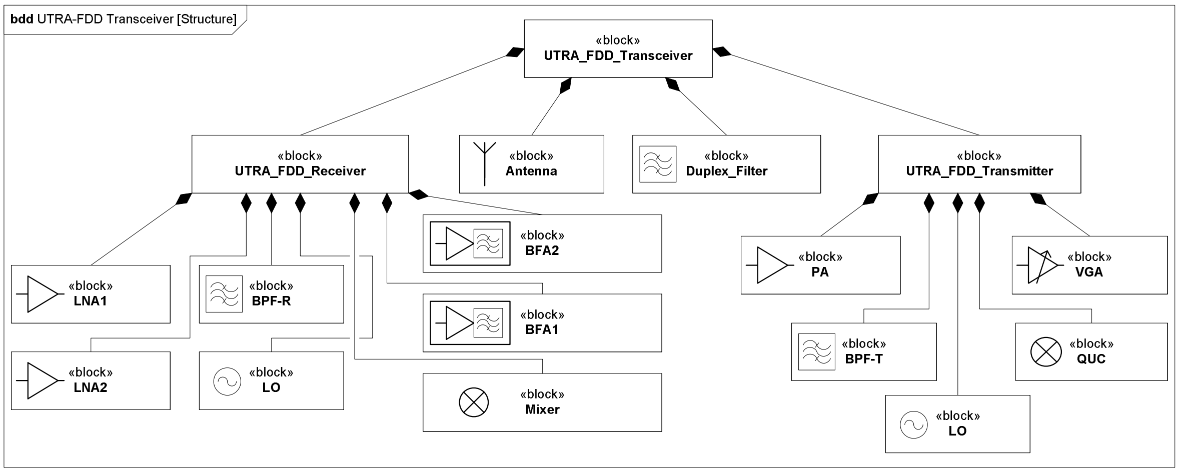


Fig. 10. An example of block definition diagram incorporating notations belonging to RF engineering domain.

Additionally, on the verification and validation level, though not illustrated in our case study, SysML allows the representation of flows, the choice of their type and the corresponding ports such that the designer can model the signal flow of the RF system and carry out an automatic check of the model leading to automated verification and validation (AV&V) of RF systems, which would be of great use in RF engineering.

Another equally useful SysML concept for RF design is the parametric diagram since RF design relies heavily on mathematical models with multiple parameters. We believe that the parametric diagram can help in building models for customized RF components and systems.

Finally, one can observe that SysML is a new language that surely needs more refinement and revision. This said, one cannot deny the importance and the consistency of the modeling concepts it presents. SysML can help RF designers to automate at least some design tasks. However, only few tools currently truly support SysML. Furthermore, the majority of them are either not sufficiently mature or were originally designed to support UML. This situation hinders significantly the widespread adoption of SysML.

5 Conclusion

SysML is a modeling language recently standardized by the OMG. It was introduced in order to address modeling issues in systems engineering. In this paper, we investigated the possibility to use this language to model RF front-ends. We first discussed the scope of UML and the emerging SysML. Then, we studied how a typical RF front-end such a UTRA/FDD transceiver can be modeled using the latter. We finally discussed the provision and the limitations of SysML to RF systems design.

This work allows us to conclude that SysML is useful in RF front-ends' design. In fact, modeling RF systems using tools that implement this language can provide significant flexibility to designers because it allows the abstraction of certain RF subsystems. Such tools can also help automating some design tasks, especially the coherence verification of the model and even the validation of its resulting implementations. This modeling approach can also enhance productivity because it captures the requirements and the constraints imposed to the system. However, great efforts must be deployed in order to enhance the SysML-supporting tools and ensure their widespread acceptance in various engineering fields.

References

1. Object Management Group: Unified Modeling Language Superstructure. Specification V 2.1.2, (2007).
2. Object Management Group: Systems Modeling Language (SysML). Specification V 1.0 (2007)
3. Belloir, N., *et al.*: Utilisation de SysML pour la modélisation des réseaux de capteurs: 14ème Colloque International sur les Langages et Modèles à Objets. Montréal, Canada (2008)
4. Vanderperren, Y., Dehaene, W.: UML 2 and SysML: An Approach to Deal with Complexity in SoC/NoC: Design, Design, Automation and Test in Europe. pp. 716 – 717 (2005)
5. Booch, G., Rambaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. pp. 10 – 12. Addison Wesley (1998)
6. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. Morgan Kauffmann OMG Press (2008)

7. Medvidovic, N., *et al.*: Modeling Software Architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology, vol. 11, no. 1, pp. 2 – 57 (2002)
8. Lange, C. F. J.: Chaudron, M. R. V., Muskens, J.: In Practice: UML Software Architecture and Design Description, IEEE Software Journal, pp. 40 – 46 (2006)
9. What are the benefits of using SysML?, www.embeddedplus.com/SysML.php
10. What is the relationship between SysML and UML?, www.sysmlforum.com/FAQ.htm
11. Madsen, P., *et al.*: UTRA/FDD RF Transceiver Requirements, Wireless Personal Communications Magazine, pp. 55 – 66 (2002)
12. Third Generation Partnership Project (3GPP): UE Radio Transmission and Reception (FDD), Technical specification 25.101 v.4.0.0 (2001)
13. W-CDMA/UMTS, FDD Technical Summary, www.umtsworld.com/technology/wcdma.htm

From high-level modelling of time in MARTE to real-time scheduling analysis

Marie-Agnès Peraldi-Frati[1], Yves Sorel[2],

[1] I3S, UNSA,CNRS/INRIA, Sophia Antipolis Méditerranée, B.P. 93, 06902 Sophia Antipolis, France.

[2] INRIA, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.

map@unice.fr, Yves.Sorel@inria.fr

Abstract. An important challenge in the domain of automotive control design is to provide a seamless flow for modelling conjointly with the behaviour, the temporal characteristics and the timing constraints of a system at different abstraction levels. In addition, this flow should provide analysis phases for validating the real-time behaviour of the functional models in regard to these constraints and a specific execution platform, To achieve this goal, we adopt a model-based approach, based on the UML MARTE profile [1], that allows the modelling of a system, with a separation of concerns between the software (functional model) and the execution platform resources (non-functional model) as well as the timing constraints (non-functional model). The temporal characteristics (offset, period) and timing constraints (deadline) are modelled with a high level notion of time called the logical time at the functional level down to the physical time called the chronometric time at the implementation level. From these high-level models we extract the temporal characteristics and the timing constraints of the application relatively to the execution platform, and we apply scheduling analysis techniques in order to provide an implementation which satisfies the timing constraints.

Keywords: UML, MARTE, Methodology, Time modelling, Scheduling Analysis,

1 Introduction

The ever increasing complexity of real-time embedded systems raises the problem of merging inside the development process, different concepts and techniques coming from the domain of software development and others concepts developed for real-time systems design.

UML (Unified Modelling Language) [2] and its domain specific extensions are becoming accepted notations to cope with the design of complex automotive real-time embedded applications.

The recent OMG standardization of the MARTE profile (Modelling and Analysis of Real-Time and Embedded systems) is an important step for modelling non-functional characteristics of real-time embedded systems. An important contribution of MARTE lies in its time model, centred on the notion of multiform time (logical or chronometric), that enriches UML with explicit time model elements (clocks, clocks type …). This is a real advance in regard to the SPT [3] (Scheduling Performance and Time) profile which considers time as an implicit notion closed to the physical time, modelled by a simple annotation onto UML models. In addition, MARTE provides two other models for describing execution platforms, and the allocation of application functions onto the resources of the platform..

Scheduling analyses are based on well founded theory widely used in the domain of real-time control systems. Applying such analysis techniques to a UML design requires extracting from functional and non-functional models the temporal characteristics of every function of these models. The main characteristics are the period, the deadline possibly equal to the period. Analysis techniques need also, as input, the Worst Case Execution Time of the function (WCET). The WCET of a function depends on its execution platform.

In this paper we shall not focus on the transformation of these temporally characterized functions into tasks which depends on the Real-time Operating System (RTOS). Later on, we shall use the term task, which is more usual in the real-time community, instead of function as soon as we shall speak of these temporally characterized functions.

The UML profile MARTE is an interesting answer to the problem of a specialization of UML for real-time embedded systems modelling.

In our approach, we use MARTE for building four models. One for representing the *functional* part of the system mainly composed of activity diagrams, state machine and structure diagrams. Another model called the *time* model contains the main time entities (clocks, clock type, clock constraints) as units for expressing the non-functional temporal characteristics (period) and constraints (deadline). Two other models represent the *resources* and the *allocation* of the functional parts onto these resources. The allocation model allows in particular, an identification of the tasks of the system by exploiting, in addition to non-functional temporal characteristics, other temporal characteristics such as the WCET which actually is hardware dependent.

The time model is composed of multiple clocks which are the inputs of a scheduling analysis of the tasks deduced from the application. As these clocks can be of different nature (logical or chronometric), the model integrates *clock constraints* for merging the clocks and for obtaining a common notion of time in order to verify the schedulability of these tasks according to a specific execution platform.

These complementary models capture different views of an application. Their relationships are based on the semantic of MARTE. This profile is recent but an experimental implementation of the profile is available in the UML editor Papyrus [4]. As we use this editor, we obtain an intermediate format which can be transformed into an input format whose syntax must be compliant with a scheduling analysis tool,

such as Cheddar [5], SynDEx [6], etc. These model transformations are out of the scope of the work presented here.
In this paper, we discuss and illustrate some methodological aspects on using MARTE at the different steps of a modelling approach that integrates both functional and non-functional modelling. Another contribution is to show how it is possible to exploit these models by extracting the temporal information and the implementation characteristics in order to provide a schedulability analysis.
The paper is organized as follows. The section 2 gives an overview of the capabilities of UML and ADL-based languages for modelling time. The section 3 presents the concepts defined in the time model of MARTE. We show how to build a time model by using the clock and the time concepts of MARTE and how to express in a non-ambiguous manner the time units and relations between the clocks of the system. We establish the link between the clock and a functional description. We illustrate the usage of the profile with the example of an ignition control system.
The section 4 presents the transition from a high level notion of time (logical time) to the real-time (chronometric time). We call this step the refinement of time which is performed by integrating and resolving the constraints between clocks and by considering the resources of the intended platform.
The last section is devoted to the exploitation of the models in order to apply a scheduling analysis. The result of such analysis is a starting point for a manual or automatic implementation solution of the functional model onto the execution platform model.

2 High level modelling of time

The adoption of a model based design approach conjointly with UML and ADL (Architecture Description Languages) is becoming promising for real-time embedded systems design [7], especially in the automotive domain. In such domain, a critical issue is to develop separate views of the software independent from the execution platform and to provide mechanisms for the projection/allocation of software onto this execution platform. In addition, both models must be endowed with timing characteristics (functional and non-functional properties).

2.1 UML and time

Applying a model based design approach to real-time embedded system design requires considering the modelling of time. UML is a modelling language for specifying, visualizing, constructing, and documenting software systems and business processes. Several profiles have been standardized by the OMG to cope with real-time.
A first profile called SPT (Scheduling Performance and Time) [3] extends UML1.4 and allows the expression of quantitative temporal information onto structural and behavioural UML models such as activity diagrams, sequence diagrams, events, and structure. This information is defined as instants, duration and observations. They

referred to an implicit notion of time so the semantic of this “implicit” time is not defined.
UML has integrated these concepts in the release 2.0. After that, a new RPF (Request For Proposal) for a UML profile focusing on embedded real-time system modelling has been proposed by the OMG. The result of the RFP heavy process is an official OMG profile called MARTE [1].
MARTE is a UML profile composed of different packages which provide adapted concepts for modelling and analyzing a real-time embedded system. Among these packages, the timing model of MARTE provides a concrete representation of clocks into UML model and, in that way, the access to duration, deadline, period and observation linked to these clocks. Another major concept introduced in the timing model is the different *nature* of clocks dense (the physical continuous time) or discrete (a discretized view of this time) and their different *types*, i.e.: chronometric (concept closed to the physical time) or logical (any repetitive event can be modelled as a clock). These different concepts are presented in the domain view chapter of the MARTE document. The implementation and the usage of these concepts is the profile itself which relies on the UML view. The user of the profile is mostly concerned by the UML view.

2.2 EAST-ADL and Autosar

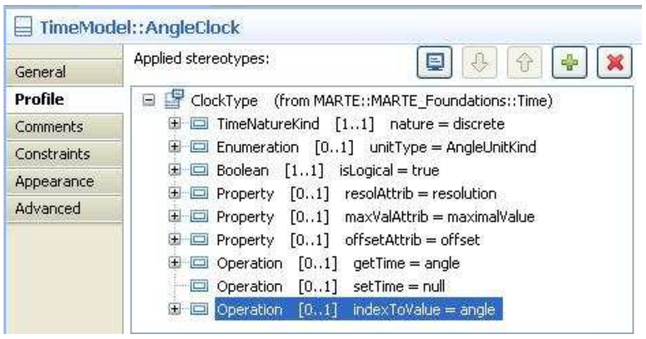
Architecture Description Languages has been adopted in the automotive domain. EAST-ADL [8] and Autosar [9] are respectively a language and the standard that allows a separation of concerns between the functional view of an application, and the non-functional ones (execution platform, environment and implementation views). This separation of concerns is compatible with a Model Based Design approach, [7][10].
Another orthogonal concept of these ADL is the decomposition of a design in domain-oriented modelling levels. To that aim, EAST-ADL provides specific model elements for catching vehicle features description, control/command modelling, software-oriented design, whereas Autosar covers the implementation level.
EAST-ADL and Autosar focus on a structural description of a system. The behavioural parts are differed to external formalisms (native languages such as Matlab/Simulink diagrams or C code…). The temporal characterization of the structural parts is limited to the expression of requirements associated with ADLFunctionType/Prototypes which are the elements for modelling the structure in EAST_ADL and Autosar software components. Such modelling of temporal aspects is not sufficient for applying timing analysis onto these models. A scheduling analysis requires a precise association of *temporal characteristics* (period and WCET) and *constraints* (deadline) to the different functionalities. Worst Case Execution Time of computations or communications between functions depends on the execution platform resources such as the CPU, the network, the bus, etc. Results on the transformation of AADL models for scheduling analysis are presented in [11]
Projects such as ATESST [12] and TIMMO [13][14] consider the problem of extending EAST-ADL2 and Autosar with MARTE temporal features.

3 Principles of Time modelling with MARTE

We adopt the recent UML-MARTE profile and particularly its timing model [15] to capture the various forms of repetitive event that trigger the functional parts of an automotive embedded system, and to model them at a high level of abstraction. As it is shown in section 3.1, the rotation of the camshaft is the main trigger of the ignition control system. In the initial requirements of the ignition control system, the value of the period and the deadline are expressed with the unit camshaft degree.
The semantic attached to these multiform time (clocks, period, deadline, jitters…) makes it possible to transform this high level model of time to the real time.

3.1 Clocks as model elements

In MARTE, time is modelled with clocks. A `clock` is an instance of a `clockType` (see Figure 1) which has multiple attributes such as the nature of the clock (discrete or dense), the unitType that is an enumeration of the different units, and the relationships between them. For instance, the relationship between *s* and *ms* is a conversion factor of 0.001. The boolean attribute isLogical indicates whether the clock is endowed with a classical unit such as *chronometric* unit (closed to the classical notion of time -UTC time) or with *logical* units (concepts of time related to logical events). A `clockType` has also resolAttrib, maxValAttrib and offsetAttrib attributes. The resolution indicates the granularity of the clock (minimal interval between two values). The maximal value corresponds to a modulo-value and the offset denotes a temporal shift at the first instant. The different operations (*getTime*, *setTime*, *indexToValue*) are defined to access the clock.



```
ClockType   (from MARTE::MARTE_Foundations::Time)
  TimeNatureKind   [1..1]   nature = discrete
  Enumeration   [0..1]   unitType = AngleUnitKind
  Boolean   [1..1]   isLogical = true
  Property   [0..1]   resolAttrib = resolution
  Property   [0..1]   maxValAttrib = maximalValue
  Property   [0..1]   offsetAttrib = offset
  Operation   [0..1]   getTime = angle
  Operation   [0..1]   setTime = null
  Operation   [0..1]   indexToValue = angle
```

Figure 1: ClockType stereotype in MARTE

A clock with a logical unit may represent any repetitive logical event in a system. Intervals between two consecutive occurrences of these events denote the instants (ticks) of the clock. Intervals are not necessarily equal. A duration represents the number of ticks between two events linked to this clock.
An example of such a logical clock is the camshaft revolution in an automotive engine. The different instants of this clock are linked to teeth detection onto the axis. The temporal distance between two consecutive teeth may vary and depends on the

engine rotation speed. This means that a logical clock is not necessarily periodic; nevertheless, it is possible to quantify a duration based on this clock, which represents the duration between two ticks on this clock. The unitType of this clock is the angle degree.
The Figure 2 shows an example of a time model with two *clockTypes*: MyIdealClockType (chronometric) and AngleClock (logical) and their *UnitKind* (MyTimeUnitKind and AngleUnitKind). Three instances of clocks are defined in the model (crkClk, camClk and myIdealClock) with different properties. We consider that the clock myIdealClock inherits, from its type, the default values for the resolution and offset. The crkClk and camClk have the same clockType but the maxValue attribute for their periods are different. One tick on camClk corresponds to two ticks on crkClk.

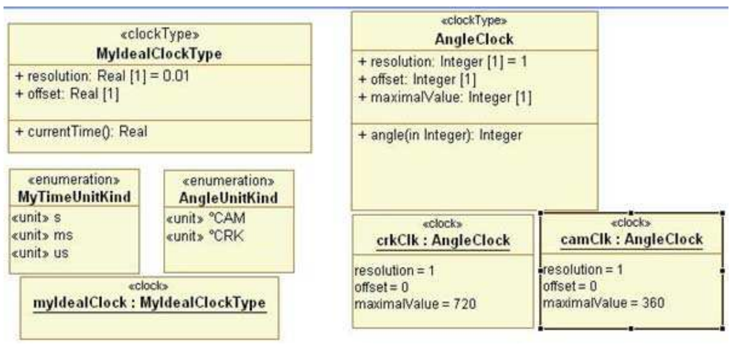


Figure 2: Elements of the Time Model

3.2 Relations and constraints between Clocks

As multiple heterogeneous clocks may be necessary in a model, MARTE allows the expression of relations between them. These relations are expressed by the way of `clockConstraint` expressions. In the MARTE profile, a specific language called CCSL (Clock Constraint Specification Language) [16] has been defined to express the different relationships between clocks.
From these constraint expressions, an automatic analysis based on partial order calculus can be applied that provides a solution for merging the clocks onto a common on considered as the reference. We call this phase the refinement of clocks that consist in the "projection" of the instants of logical and heterogeneous clocks onto a common one. In our example the relation between the crkClk and camClk clocks is given by the following CCSL expression:

```
    camClk = crkClk filteredby 0b(10)
```

This means that camClk is a subclock of crkClk, and that the ticks of the camClk are coincident with the ticks of the crkClk when applying the binary mask 10 as a filter.

For a given application, the analysis of a set of clock constraints provides a solution, if this solution exists, of the interleaving of these clocks.
Clock constraints can be also used simply to transform a duration from one unit to another one. In the case of the crkClk, the relation between a second (s) and a °CRK depends on the rotation speed of the Crankshaft (Rotation Per Minute). Equation 1 expresses this relation.

$$1_{\text{°CRK}} = \frac{1}{RPM \times 6} s\,. \tag{1}$$

3.3 UML Behaviour modelling using Clocks

A MARTE model containing clocks makes it possible to associate instant (occurrences) to events, or intervals (durations) to behaviours of a UML model. To achieve this goal, UML behaviours can be stereotyped with `timedProcessing`.

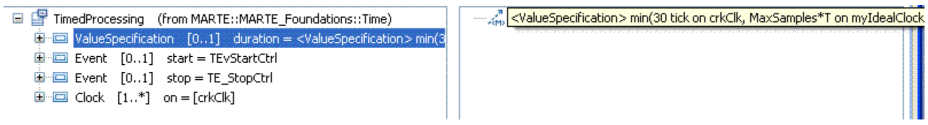

```
TimedProcessing   (from MARTE::MARTE_Foundations::Time)
    ValueSpecification   [0..1]   duration = <ValueSpecification> min(3
    Event   [0..1]   start = TEvStartCtrl
    Event   [0..1]   stop = TE_StopCtrl
    Clock   [1..*]   on = [crkClk]
```
```
<ValueSpecification> min(30 tick on crkClk, MaxSamples*T on myIdealClock
```

Figure 3: timedProcessing attributes

A timedProcessing may be any UML behaviour such as an activity diagram, a sequence diagram, a state machine and is linked to a clock (*on* attribute).
The main effect of such association is the possibility offered to the designer for expressing temporal characteristics and constraints onto behaviours. In MARTE these constraints are *timedValueSpecification* which represents either a duration or an interval, in between which, the behaviour should execute.
The Figure 3 shows the attributes of a TimedProcessing stereotype. The *clock* attribute indicates the reference clock on which the durations and events are measured. Two significant *timedEvents* can be associated with the start and stop of this behaviour. The *start* attribute indicates the event that triggers the behaviour. The *stop* attribute is an event produced by the behaviour at the end of its execution. This event materializes the duration of the activity and can be used in a *timedValueSpecification* to express a temporal constraint.
Of course, multiple factors influence the temporal distance between these two events. An advantage of using these events is the possibility of reasoning on the end of a behaviour independently to its physical execution support.
A *timedValueSpecification* expresses a constraint onto the activity duration. Notice that different clocks can be involved in a duration expression. In the equation 2 extracted from the paper [17], the duration is expressed as follows:

$$\text{duration} \prec MIN\begin{pmatrix} \text{30 tick on crkClk,} \\ \text{MaxSamples* T on MyIdealClock} \end{pmatrix} \tag{2}$$

The duration depends on two values measured onto two different clocks. Duration must be less than the minimum between 30 tick measured onto the crkClk clock and MaxSamples*T tick measured onto myIdealClock. MaxSamples is a variable of the application specification.
This equation can be solved after integrating the clocks constraints as explained in the paragraph 3.2. In this case, it consists in translating the crkClk unit to the second unit.

3.4 Illustration on an ignition control system

We illustrate the MARTE Time Model usage on the example of an automotive engine control system. We focus on a particular part of this control that is the correction of the ignition. The activity diagram on Figure 4 shows that the ignition of the spark plug in an engine depends on different corrections imposed by physical phenomenon such as the knock, the temperature variation and the warm-up of the engine. Multiple sensors and actuators participate to this correction. The correction controllers must be executed periodically and their executions may overlap. The period unit is the crkClk clock. The period, the offset and the deadline of these actions are also linked to this logical time base.

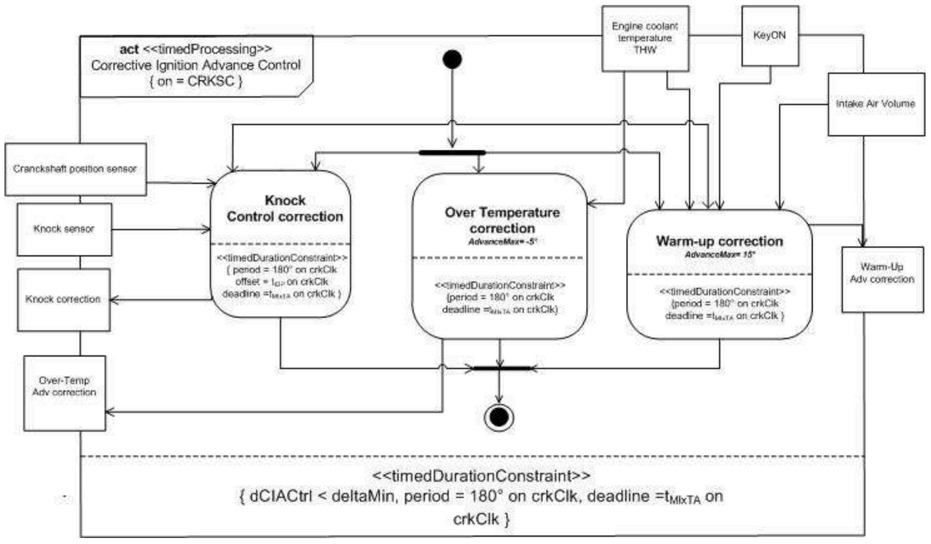

act <<timedProcessing>>
Corrective Ignition Advance Control
{ on = CRKSC }
Engine coolant temperature THW
KeyON
Intake Air Volume
Crankshaft position sensor
Knock sensor
Knock correction
Over-Temp Adv correction
Warm-Up Adv correction
Knock Control correction
<<timedDurationConstraint>> { period = 180° on crkClk offset = t_{IGP} on crkClk deadline = t_{MIxTA} on crkClk }
Over Temperature correction
AdvanceMax= -5°
<<timedDurationConstraint>> {period = 180° on crkClk deadline = t_{MIxTA} on crkClk}
Warm-up correction
AdvanceMax= 15°
<<timedDurationConstraint>> {period = 180° on crkClk deadline = t_{MIxTA} on crkClk }
<<timedDurationConstraint>>
{ dCIACtrl < deltaMin, period = 180° on crkClk, deadline = t_{MIxTA} on crkClk }

Figure 4: Correction activity of a control engine system

We associate to each correction controller a temporal characteristic such as its period and a timing constraint such as a deadline. For the sake of visibility we have represented these constraints at the bottom parts of the actions and the activity. In an actual model using tools, these constraints are modelled with timedValueSpecification expressions which are elements of the design.
A consequence of the activity diagram hierarchy is that time constraints can be expressed at the different levels of this hierarchy. Thus, time constraints at the lower levels must comply with those of the higher level.

A verification of such constraints can be done by applying mathematical rules and reasoning on a high level of abstraction of time. At this level we can obtain results about the feasibility of this control according to the different constraints and some other characteristics of the system.
Examples of the feasibility results are: all the correction controls can be executed on time in the case of a 4-stroke engine with a maximum engine rotation speed of 4000 rpm but cannot be executed if the rotation speed exceeds this value. These results, detailed in the paper [17], were obtained by a manual calculus applied to functions executing sequentially (non preemptive cyclic scheduling). On the example presented figure 4 multi-task scheduling is intended. For that purpose the tasks has been enriched with the temporal characteristics requested for this type of analysis.

4 From logical to physical real-time

4.1 Clock refinement

The modelling of multiple heterogeneous clocks in the time model allows a high level modelling of temporal aspects of a system. The goal is to apply scheduling analysis onto these models. Such analysis requires two conditions. The first one is, at the behavioural modelling level, the necessity to cope with a common notion of time. The second condition is the integration of the execution platform model and the allocation constraints in order to integrate the physical time (the one of the CPU processor).
The paragraph 4.2 addresses this second aspect. Concerning the common notion of time, it is obtained at the behavioural model level by the integration of temporal constraints. Equation 1 gives the relation between the crkClk and the idealClock (s). The parameter *RPM* represents the engine rotation speed. This parameter depends on the automotive phases (acceleration, deceleration …). To evaluate the schedulability of the different tasks, we have to consider the worst case execution time corresponding to the maximum value of the RPM (MAXRPM=4500). By considering this value, we may use the clock constraint expressed in Equation 1 to transform the period values which time unit is the crankshaft degree °CRK to period expressed in second. The Table 1 gives the result of such transformation.

4.2 Integration of the execution platform characteristics

In a MDE design process, the execution platform model is built separately from the application model. The MARTE profile provides a set of resources model elements that supports the modelling of the application. The association between an application model and an execution platform model is obtained through an allocation model.
Figure 5 shows a structural description of the architecture of the ECU on which the engine controller will be executed. The execution platform is modelled as a set of microcontrollers and a memory; all the elements are interconnected by a bus.

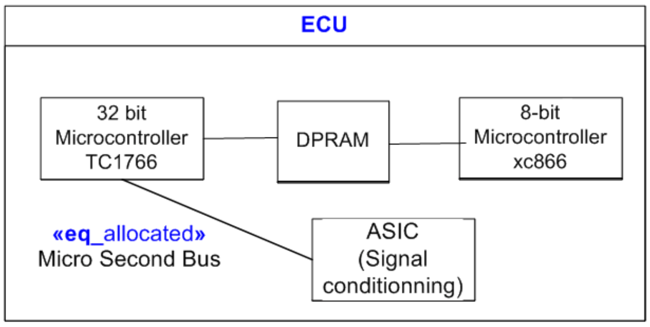

ECU
32 bit Microcontroller TC1766
DPRAM
8-bit Microcontroller xc866
«eq_allocated» Micro Second Bus
ASIC (Signal conditionning)

Figure 5: ECU execution platform

The execution platform can be annotated with temporal information based on logical or chronometric clocks. Figure 6 provides a timed-view of the different elements of the execution platform. A computation resource can be seen as a clock with a resolution that corresponds to the duration for executing one instruction cycle on this processor. As the Infineon TC1766 is a PCP2 (single cycle instruction), it means that one instruction on this processor takes a duration of 125 10^{-10} second. The temporal characteristic of a bus is generally its transmission rate.

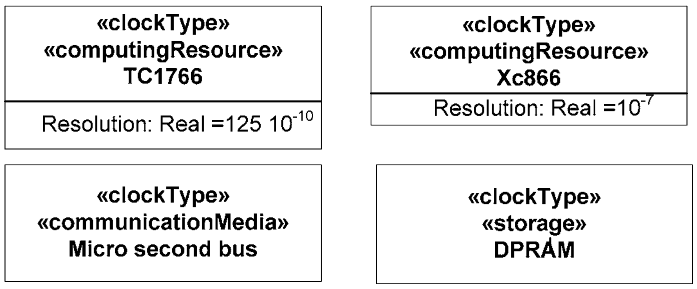

«clockType» «computingResource» TC1766
Resolution: Real =125 10^-10
«clockType» «computingResource» Xc866
Resolution: Real =10^-7
«clockType» «communicationMedia» Micro second bus
«clockType» «storage» DPRAM

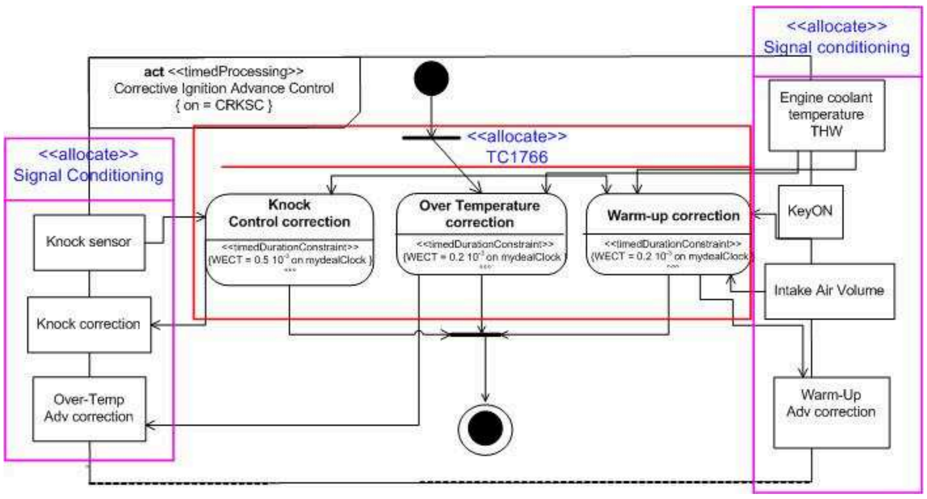
Figure 6: Time model of hardware

This abstract view of the execution platform can be enriched with others similar non-functional characteristics such as the memory cost, the power consumption. Such model allows a verification of the execution platform capabilities in regard to the constraints and the characteristics of the application parts.

4.3 Allocation of functionalities onto execution platform resources

The association between the application model and the execution platform model is expressed by the allocation model. An allocation associates every element of the application model to a resource of the execution platform model. In MARTE allocations, are annotated with time constraints and temporal characteristics. The allocation modelling establishes the link between the model views of time and the actual physical time.

Allocations can map structural to structural elements, and behavioural to behavioural or structural elements. The Figure 7 represents the potential allocations of sensors, actuators and actions of Figure 4 onto the possible computation resources of the ECU. The sensors and actuators behavioural parts are allocated to the ASIC (signalConditioning). The functions are allocated to the Infineon TC1766. In some cases, a behavioural part may have several possible allocations onto different resources. The allocation of a function onto a physical resource implies a temporal cost (the WCET). This cost is a new information that appears on the allocation diagram at the bottom part of each action.



<<allocate>>
Signal conditioning
act <<timedProcessing>>
Corrective Ignition Advance Control
{ on = CRKSC }
<<allocate>>
TC1766
<<allocate>>
Signal Conditioning
Engine coolant temperature THW
KeyON
Knock sensor
Knock Control correction
<<timedDurationConstraint>> {WECT = 0.5 10⁻³ on mydealClock }
Over Temperature correction
<<timedDurationConstraint>> {WECT = 0.2 10⁻³ on mydealClock }
Warm-up correction
<<timedDurationConstraint>> {WECT = 0.2 10⁻³ on mydealClock }
Intake Air Volume
Knock correction
Over-Temp Adv correction
Warm-Up Adv correction

Figure 7: Partition of activities

Table 1 is a summary of the different temporal characteristics of the behavioural parts. The period, offset and deadline values have been extracted from the behavioural models.
The period expressed initially with the time unit °CRK has been translated in second by applying the time constraint relation on equation 1.
The offset and deadline were expressed as relative dates in the behavioural model with the time unit °CRK. We give in Table 1 the actual values of these parameters obtained by calculating the corresponding values in milliseconds. The last row of the table represents the duration of the deadline which depends on the offset and the deadline date.

	Knock	**Over Temp**	**Wam-up**
WCET (on TC1766 in ms)	0,5	0,2	0,2
Period(°CRK - *ms*)	180 - *6,66*	180 - *6,66*	180 - *6,66*
Offset date (°CRK - *ms*)	24 – *0,888*	0 - *0*	0 - *0*
Deadline date (°CRK / *ms*)	50 – *1,851*	50 – *1,851*	50 – *1,851*
Deadline duration (°CRK / *ms*)	26 – *0,962*	50 – *1,851*	50 – *1,851*

Table 1: Temporal information associated to tasks

The WCETs are obtained either by emulating or profiling the tasks corresponding to the correction controllers represented in the behavioural model. The time base in this case is the time base of the processor on which these tasks will execute, i.e. the

physical clock. The value of the deadline has been extracted from the book [18] which gives some actual parameters values of an ignition engine controller.
The next step consists in exploiting these characteristics in the scheduling analysis phase.

5 Scheduling analysis

5.1 Principles

The scheduling analysis may start as soon as the application models with its associated timing model, and the architecture model, are available. Mainly, it consists in exploiting the timing information, i.e. the temporal characteristics (periods and WCETs, the latter depending on the allocation) as well as the timing constraints (deadlines), attached to each temporally characterized function of the application model. The allocation model allows the designer to determine the actual timing characteristics which vary with the various possible computing resources each application element can be allocated to. Multiple potential allocations can be considered for a function when several computing resources are able to implement it. In our case the computing resource is unique since we address uniprocessor architecture. Nevertheless several versions of this unique processor may be considered. In this case the schedulability analysis must be iterated according to the different processors which induce different timing values of the WCET.
The processor of the execution platform is supposed to provide a RTOS, e.g. OSEK in the case of our example. This RTOS is the standard for the automotive domain. The schedulability analysis assumes also that the given RTOS supports the scheduling policy the analysis is based on. This will guarantee that the real-time behaviour of the application, running on the chosen architecture, will satisfy the real-time constraints.
Actually, this assumes that the WCET were carefully determined and enough margins were taken to approximate the cost of the RTOS itself, i.e. the cost of the scheduler including the cost of the preemption if it is allowed by the scheduling policy.

5.2 Illustration on an ignition controller

In the ignition control system example, the three behaviours: Knock control correction, Over temperature correction, and Warm-up correction are associated to a couple of temporal characteristics (deadline, period), and a WECT that was determined according to the execution platform resources it was allocated to. These associations lead to a system of three periodic tasks: Knock, OverTemp, and Warm-up. Their periods initially expressed in °CRK have been translated in milliseconds (from the idealClock) by applying the time constraints between the two clocks. The resulting values are listed in Table 1.

With these data, it is possible to perform a scheduling analysis based on a fixed priority scheduling policy. Every tasks is periodic, has a WCET, and a deadline. In addition, preemption is allowed. As the deadline of each tasks is less than its periods, the system of tasks can be scheduled according to the Deadline Monotonic (DM) scheduling algorithm [19], i.e. the task with the smallest deadline has the highest priority, assuming these tasks are independent. If they are not independent a more complex schedulability analysis must be performed, but that does not change anything to the proposed approach. We do not focus on the schedulability analysis itself but on the way it is possible to perform it from the previous models, manually or possibly automatically. In this context, the scheduling analysis using DM algorithm amounts to verify the following sufficient condition. As mentioned before, to be consistent the RTOS running on the considered uniprocessor must use also this algorithm.
The system is schedulable if:

$$\sum_{i=1}^{n} \frac{WCET_i}{Deadline_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \text{with } Deadline_i \leq Period_i \qquad (3)$$

We chose here the DM fixed priority policy instead of the Earliest Deadline First (EDF) variable priority policy because the scheduler is simpler, and thus its cost is easier to approximate. This approximation is a fundamental hypothesis used in the DM schedulability analysis. Usually the industrial designers take a margin equal up to 30% of the task WCET. With these assumptions our automotive example with the three tasks mentioned above is schedulable if:

$$\frac{WCET_{KC}}{deadline_{KC}} + \frac{WCET_{OT}}{deadline_{OT}} + \frac{WCET_{WU}}{deadline_{WU}} \leq 3(2^{\frac{1}{3}} - 1) = 0.779 \qquad (4)$$

Considering the values of Table 1 we can conclude that this system of three tasks is schedulable with the assumption of a MaxRPM equal to 4500. In this case, the left part of the equation is equal to 0.735 < 0.779. On the other hand, if we consider a MaxRPM equal to 6000, the left part of the equation is equal to 0.980 > 0.779 and the system of tasks is not schedulable.
Another constraint to be verified is the timedValueSpecification of the activity diagram (CorrectionAdvanceControl). According to the timedDurationConstraints expressed on Figure 4 the equation 5 must be verified.

$$t_{IDP} + WCET_{KC} + WCET_{OT} + WCET_{WU} \leq t_{MIxTA} \qquad (5)$$

This equation is also valid.

6 Conclusion

In order to cope with the complexity of real-time embedded systems, the Model Based Design approach promotes a separation of concerns between the model of the application (functions) and the model of the execution platform.
UML and its profiles are largely used to model both parts. The recent standardization of the UML MARTE profile extends UML with temporal information and physical resources modelling capabilities. Applying this profile to a model based design makes it possible to enrich the application and execution platform models with precise and semantically well founded temporal information. As this information corresponds to explicit model elements endowed with a clear semantic, they can be extracted from the model. Consequently, MARTE models can be used as a starting point for methods and tools intended for schedulability analysis. They take into account temporal information and timing constraints for verifying deadline constraints.
In this paper, we presented the MARTE model elements associated with the time model package of MARTE, and we illustrated their uses on an automotive case study. Four models were presented, the *functional* model, the *time* model, the *allocation* model, and the *execution platform* model. While temporal information (periods) and constraints (deadlines) are associated with the functional model and are independent from the execution platform model, other timing information (WCET) are dependent of the platform, i.e. the computing resources.
In this approach, there are two notions of time, the time linked to the functional model logical time and the physical time related to both the allocation model and the execution platform model. We showed how to establish the link between logical time and physical time through the allocation model.
From these models we extracted the physical timing information and used them to straightforwardly perform a schedulability analysis. We use an analysis based on the DM algorithm, but others types of analyses are possible, which concludes that the illustrative application, characterized with the temporal characteristics and constraints, is schedulable onto the given execution platform.

References

1 MARTE OMG Specification. *A UML Profile for MARTE, Beta 1*.OMG Adopted specification ptc/07-08-04. August 2007.
2 OMG. *UML 2.1 Superstructure Specification*, April 2006.OMG document number: ptc/2006-04-02.
3 OMG. *UML Profile for Schedulability, Performance, andTime Specification*, January 2005. OMG document number: formal/05-01-02 (v1.1).
4 *Papyrus*: graphical UML2 modelling editor, http://papyrusuml.org
5 *Cheddar*: http://beru.univ.brest.fr/~singhoff/cheddar
6 *SynDEx*: http://www-rocq.inria.fr/syndex/
7 T. Schattkowsky, W. Muller, *Model-based design of embedded systems*, IEEE symposium on Object-Oriented real-time distributed computing, , pp113-128, Vienna May 2004
8 ITEA project. *EAST-ADL: The EAST-EEA Architecture Description Language*, June 2004. ITEA Project Version 1.02.

9 *Autosar* specifications, www.autosar.org
10 H. Gronniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, B. Rumpe, *View-centric Modeling of Automotive Logical Architecture*, Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV. Informatik-Bericht 2008-02, CFG-Fakultät, TU Braunschweig, 2008.
11 O Sokolky, I. Lee and D. Clarke, *Schedulability analysis of AADL models*, 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'06), Island of Rhodes, Greece, April25-26, 2006.
12 ATESST, *ATESST project reports of the Advancing Traffic Efficiency and Safety thought Software Technology* (ATESST) project. *http://www.atesst.org*.Final versions, 2007
13 *TIMMO* project, www.timmo.org
14 Oliver Scheickl, Michael Rudorfer , *Automotive Real-Time Development Using Timing-augmented AUTOSAR Specification*, BMW Car IT, Proc. of the 4th European Congress Embedded Real-Time Software (ERTS 2008), Toulouse, France, January 29 - February 1, 2008
15 Ch. André, F. Mallet, and R. de Simone, *Modeling Time(s).* In MoDELS'07, 10th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems. LNCS 4735, 2007.
16 *CCSL Clock Constraint Specification Language*, MARTE OMG Specification [1] Annex C3 pp419-430.
17 F. Mallet, C. André M-A. Peraldi-Frati *Multiform time in UML for Real-Time Embedded Applications*, 13th IEEE Inter. Conf. on Embedded and Real-Time Computing Systems and Applications, Daegu (Korea) August 2007.
18 R. Bosch GmbH, *Automotive Handbook*, 7th edition, Bentley Publishers
19 C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, pp. 46-61, january 1973

A Reinterpretation of Patterns to Increase the Expressive Power of Model-Driven Engineering

Matteo Bordin[1], Marco Panunzio[2], Carlo Santamaria[2], and Tullio Vardanega[2]

[1] AdaCore
46 rue d'Amsterdam, 75009 Paris, France
bordin@adacore.com
[2] University of Padua, Department of Pure and Applied Mathematics
via Trieste 63, 35121 Padova, Italy
{panunzio,tullio.vardanega}@math.unipd.it

Abstract. The model-driven engineering (MDE) paradigm wishes to raise the abstraction level of the user design space, while resting on the automated generation of all lower-level artifacts. Under the MDE approach the focus of verification and validation increasingly verges on models. As a consequence, the expressive power availed to the user is often considerably restricted to ensure that the models are amenable to static analysis. Inherent tension thus arises in the very essence of MDE between the restraints to be placed for a better good on the user-level expressive power and the user need and expectation to be able to operate in a modeling space delivered of platform dependences and constraints. In this paper we contend that a new notion of modeling patterns may help resolve the conflict and increase the expressive power in the user space without jeopardizing the integrity and effectiveness of the transformation process.

1 Introduction and related work

Motivation. Model-Driven Engineering [1] aims to decrease the time and cost of software production and to increase quality, by leveraging on the factorization of best practices in programming and implementation. The MDE paradigm strives to raise the abstraction level of the user space and to generate all lower-level artifacts automatically, source code, analysis models, and documentation alike. MDE wishes to deliver the user from the burden of dealing with platform-specific implementation details, and to concentrate instead on the (platform-independent) specification of the solution. The assumption behind this vision is that the implementation may be largely if not completely delegated to the automation capabilities of platform-specific development frameworks.

At present however, the adoption of MDE for the high-integrity application domain is still a challenge. Already in the general case in fact, it may be overly difficult to provide sufficient assurance that all properties attached to the user model and validated at that level of abstraction be correctly propagated throughout model transformations, and preserved upon deployment and execution. This provision demands a level of control over and proof of the production process, which is difficult to attain for the general user.

State of the art. The most prominent efforts in modeling languages for real-time systems in the industrial landscape to date are AADL and MARTE. AADL [2] focuses on the modeling of schedulable entities, of information and control flows, and on the analysis thereof. AADL has also recently been augmented with specific annexes targeting behavioral aspects and error treatment mechanisms. AADL conveys all user concerns, such as scheduling, flow and behavioral modeling, into a *single* modeling view. The main advantage of this choice is the comparative simplicity and cohesiveness of the modeling language: the AADL syntax is particularly compact, and each semantic concept can easily and directly be expressed with a single combination of syntactic constructs. The single-view modeling of multiple concerns however limits the power of abstraction considerably and pushes it down to the implementation level. The abstraction level of AADL models thus gets downcast to that of the underlying implementation as intended by the target execution platform and the accompanying theories of analysis, which is quite contrary to the intention of MDE.

MARTE [3] is an OMG effort to bridge schedulability-oriented modeling with system-level aspects such as flow analysis and software/hardware interaction. (MARTE can in fact be used in conjunction with SysML [4].) As of September 2008 the MARTE specifications are in official beta status. MARTE suffers from the gigantism typical of several OMG standards. As in UML, a MARTE model is comprised of several views, the consistency of which is *not* assured by the underlying metamodel. Moreover, even if vastly more expressive than AADL (especially for time-related semantics), numerous syntactic constructs in MARTE insist on one and the same semantic concept and thus overload it. These characteristics make MARTE models rather complex to understand. Ultimately, the semantics expressible with MARTE is close to that assumed in common scheduling analysis theories.

While both AADL and MARTE provide platform-independent ways of modeling software systems in manners amenable to static analysis, the abstraction level of their modeling space is restrained by constraints arising from the execution semantics intended for the target platform. In fact, in both AADL and MARTE the abstraction level at PIM is almost equivalent to that at PSM. That closeness eases the preservation of model attributes and properties across model transformations, but at the cost of permitting only a shallow distance between PIM and PSM.

The last modeling language we consider is HRT-UML/RCM [5,6], the authors' own proposal, an MDE infrastructure devised in the ASSERT project [7] to defy this challenge especially. In ASSERT, HRT-UML/RCM was used for the development of an industrial-scale real-time embedded system by one of the major prime contractors in European space industry.

HRT-UML/RCM aims to: (i) provide a design environment in which the user solely operates in the PIM space, with the only exception of the specification of hardware configuration and application deployment; and (ii) support an MDE methodology characterized by principles of correctness by construction [8] and of property preservation [9], across all model transformations including at run time.

The HRT-UML/RCM model space does not allow any semantic variation points: the run-time semantics expressed in its models thus always is completely defined.

In HRT-UML/RCM, the user specification of the PIM is declarative, while the transformation process applied to it corresponds to an implementation designed to be provably correct by construction. The resulting product consequently does not need to be verified a posteriori on a per-system basis, but only requires a single per-platform validation, with important cost savings for the developer.

HRT-UML/RCM has for now elected to produce a single PSM from the PIM. space, though other PSM may in principle be generated, for instance to address dependability, safety and security concerns. In HRT-UML/RCM the PSM is also used as a *Schedulability Analysis Model* (SAM). The SAM represents the semantics of the system model to the extent of allowing static analysis of the feasibility and sensitivity of its timing behavior. The SAM generated in HRT-UML/RCM is comprised of a set of comparatively simple building blocks, which, through correct-by-construction composition, may arrive at encompassing arbitrarily complex execution semantics.

HRT-UML/RCM seamlessly integrates round-trip support for feasibility and sensitivity analysis and makes it start and end at the PIM [10] (see figure 1). While the analysis is of course made on the SAM, its results are propagated back to the PIM, which is possible as the entire model transformation logic is deterministic and reversible, hence it may be easily followed backwards.

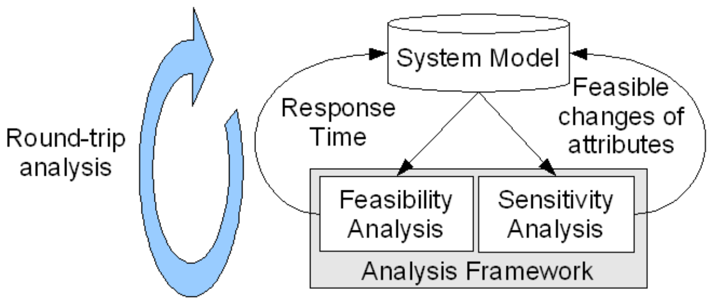


Fig. 1. Round-trip timing analysis in HRT-UML/RCM

HRT-UML/RCM also supports automated generation of source code. This leg of the transformation process starts from the SAM, which eases the provision of constructive proofs that the system at run time does correspond to what was analyzed and deemed feasible in the SAM. In the case in instance the complexity of the code generator is modest since the SAM is very close to the system at run time and the code generation engine makes extensive use of simple patterns [11].

A factor that greatly facilitated our attainment of correct-by-construction transformations was the decision to hoist the observance of the RCM constraints from the PSM space up to the PIM. This decision however has the downside that it pushes back onto the user the need to think in implementation terms (so that the RCM restrictions are not violated) in contrast with the promise of delivery from those very concerns.

Contribution. The challenge we wish to defy is to raise the user space to a higher level of abstraction (closer to the problem domain and more distant from the constraints of implementation) while guaranteeing preservation and assurance of properties across deeper model transformations. In this paper we discuss the role that *MDE patterns* may play to this end.

Previous work on the use of patterns in real-time systems [12,13] predominantly if not exclusively considered *design patterns* in the way they were promoted by the "Gang of Four" [14]. We contend that such a view fails to take full advantage of the emerging MDE paradigm. We show additional classes of potentially useful patterns, and attempt an initial classification of them from the broader perspective of the transformation space.

The remainder of the paper is structured as follows: in section 2 we define what we mean by "expressive power" in the context of MDE; in section 3 we draw a tentative classification of MDE patterns against the hierarchy of transformations implied in the process; in section 4 we discuss a few example patterns in some details and finally we draw some conclusions.

2 Expressive power in Model-Driven Engineering

We consider the expressive power of a language to relate to its economy of expressions: the more synthetic the language entities and the denser their semantic contents, the greater the expressive power. By this definition, the keywords of a programming language are much more expressive than the words in the instruction set of the target processor. In the context of programming languages the transformation of a more expressive program text into a lesser one is taken care of by the compiler. It is a well-know observations that the very existence of a compiler for a given target permits to implement other programming languages equipped with greater expressive power [15].

The expressive power thus is the capability of expressing, synthetically, high-level concepts with a finite set of language terms with known meaning and with the guarantee that they can be correctly translated into a semantically equivalent set of entities that belong to an underpinning implementation language.

This very principle lies at the heart of the MDE paradigm too. The PIM space may exploit a dictionary of technology- and implementation-independent terms, which model transformation translates to terms that belong to a specific execution platform. Source code is only one of many possible PSM produced by a model transformation of a PIM. In general it is possible to generate a set of PSM each of which serves a different purpose. Each PSM may thus represent the implementation of the PIM at a distinct level of abstraction, or viewpoint, each of which focusing only on the concerns of interest to selected view-specific stakeholder.

The expressive power of PSM is often constrained by the level of formalism required to permit the sound application of domain-specific analysis techniques. For example, constructs that incur non-determinism or unbounded execution behavior may be removed from the PSM language in the prevailing interest of facilitating static analysis.

In general, two opposite approaches can be pursued to increase the expressive power availed to the PIM space:

- granting the maximum possible (in principle, full) freedom of expression to the user and then verifying *a-posteriori* whether the user model can be successfully turned into a semantically-equivalent and legal PSM;
- capturing all of the constraints that propagate up from the PSM and striving to relax all of them for which a transformation pattern may be devised which may be proven correct *a-priori* and whose eventual overhead may be deemed acceptable to the application.

In the former approach the MDE infrastructure provides the designer with expressive power as large as the user can have. In this case however there is no guarantee that a legal PSM may be obtained from automated transformation of the user model. The verification must be therefore made a posteriori and on a per-model basis.

With the latter approach instead, the infrastructure clearly continues to restraint the expressive power, but for the benefit of a-priori guarantees that any user model in the PIM scope can be automatically transformed into a legal PSM. The metamodel is then the key element to ensure that the user model space is exclusively populated with legal entities, attributes and relations.

In HRT-UML/RCM the PIM modeling space is directly restrained by the bottom-up propagation of semantic constraints from the underlying computational model, cast onto the RCM metamodel. The RCM originates in language-neutral terms from the Ravenscar Profile of the Ada programming language [16]. The Ravenscar profile: (i) forbids the use of language constructs that may incur non-determism or unbounded execution time; (ii) only allows asynchronous one-way communications mediated by shared resources equipped with a deterministic synchronization protocols, like the Priority Ceiling Protocol (direct inter-task communication are thus prohibited). RCM further requires threads to have a single suspension and a single (cyclic or sporadic) source of activation events. All those restrictions are imposed on the PSM to ensure that all models in that space be statically analyzable. We want to be able to turn every possible user model in the PIM space into a semantically equivalent PSM, which also be correct by construction against the RCM restrictions.

When the applicable RCM constraints propagate up to directly restrict the user modeling space, the expressive power of the PIM can only be marginally larger than a simple bottom-up projection of the expressive power of the PSM (cf. figure 2.a).

HRT-UML/RCM offers a set of declarative stereotypes to increase the abstraction level in the PIM space. For example, provided services are decorated with attributes that express their intended concurrent behavior and timing properties without the user having to bother with how to implement them. Nonetheless, several restrictions of the RCM (which thus pertain to the PSM) still directly apply to the PIM. For example, the RCM forbids the creation of deferred operations with out parameters (i.e., operations executed by a server-side thread which may return values to the caller) for that would incur synchronous blocking semantics in violation of the Ravenscar restrictions.

We believe that a more advanced use of MDE patterns may help us extend the expressive power of the PIM. We want to attain the maximum possible increase while maintaining the guarantee that *all* the user models expressible in the PIM space can be represented as (arbitrarily complex and yet correct by construction) compositions of legal PSM entities. A deterministic yet efficient function must exist to transform the

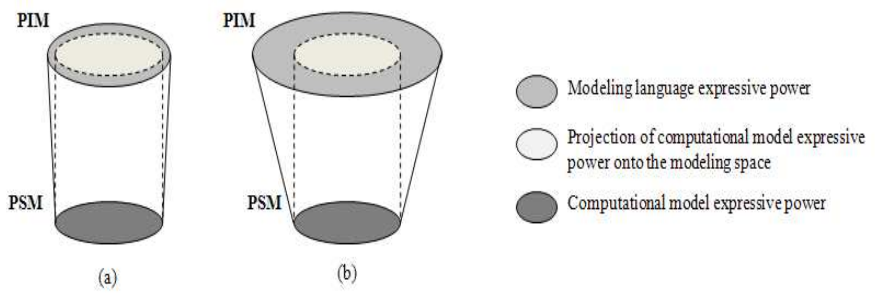

PIM
PIM
PSM
PSM
(a)
(b)
Modeling language expressive power
Projection of computational model expressive power onto the modeling space
Computational model expressive power

Fig. 2. Relation between expressive power of computational model and modeling language. Figure (a) represents a modeling space which is a projection of the underlying computational model: this is the current situation of HRT-UML/RCM. In figure (b), we represent our goal: the user-level modeling semantics is wider than the projection of the PSM computational model; but it is still possible to trace such semantics to an equivalent representation that respects the RCM constraints.

constructs that belong in the extended PIM space into a combination (thus, intuitively, a composition pattern) of those primitively present in the PSM (see figure 2.b).

Granted, the additional expressive power can only come at some cost: the larger the distance between the declarative language of the PIM and the implementation language of the PSM, the greater the time and space overhead of the model transformation, of its code products and of and its verification effort.

3 Classification of patterns

Let us first introduce a tentative classification of patterns in the MDE landscape. With this classification we maintain that MDE patterns distinguish themselves by the abstraction level at which they are applied. The abstraction level at which any given pattern is considered to belong thus becomes the central element to decide their goodness of fit to serve our objective.

In our current classification, we recognize:

– *determined patterns*: this kind of patterns are explicitly instantiated by the user, who recognizes their possible use in the current design and manually embeds them in the model as a solution to the problem at hand
– *executive patterns*: this kind of patterns are the result of the identification of features or constraints specific to the platform(s) of interest. They are directly included in the metamodel and the associated transformations, for they provide satisfactory solutions to known implementation-specific problems. These patterns are not directly available to the user, who is not aware of their use
– *declarative patterns*: this kind of patterns are solutions to recurrent problems in the application domain. The design infrastructure is aware of these patterns, which are offered to the user as built-in components that can be safely embedded in transformations of the user model.

Determined patterns represent user-perceived solutions to recurrent problems in the modeling space of the application. The user recognizes a specific problem in the application requirements and manually augments or adapts the model to host an instantiation of the desired pattern(s). The design patterns discussed in [14] clearly fall in this category. It is important to notice that, following our definition, this kind of patterns do not increase the expressive power of the MDE infrastructure, since they are user-level constructions that result from the assembly of primitive entities. Consequently, only the latter actually "exist" in the model. The user-level constructions instead have no direct representation across model transformation since the MDE infrastructure is unaware of their existence.

Executive patterns encode traits of the implementation domain of interest. A computational model, a set of constraints on thread activations and suspensions, archetypes to factorize common behaviors, are all example of executive patterns. Executive patterns distinguish themselves because they must necessarily be encoded in the metamodel and fixed once and for all upon its creation. Since they are part of the metamodel, which constrains the legal design space, they determine the PSM expressive power. As the MDE paradigm wishes to lift the abstraction level of the user space away from implementation details, the executive patterns need not be directly visible to the user.

Declarative patterns are themselves solutions to a recurrent problem of the application domain. More specifically, the designer recognizes a known problem in the system specification and uses one of these patterns to solve it. Declared patterns must be semantically understood by the user and recognized as solutions to specific problems in the application space. They however require no implementation from the user, who just need to cause them into existence. It is the process of model transformation in fact that instantiates the required patterns and embeds them in the user model. For this reason declarative patterns must pose no obstacle for the products of model transformation to stay within a legal, correct by construction, design space. The most effective way to achieve this is for declarative patterns to exist in the metamodel and consequently be used as constructive elements of the model transformation logic. Following our definition, declarative patterns augment the expressive power availed to the user, since they result from constructs (e.g.: attributes) that model transformation resolves in a predefined and correct by construction composition of primitive entities of the PSM. Furthermore, the semantics of a declarative pattern is known and well defined and the enforcement of legal conditions for its application can be assured by the MDE infrastructure itself; both these characteristics are instead contemplated in the application of determined patterns.

One of the goals of this categorization is to let the reader appreciate how important those patterns may be to the MDE process and how much they may influence the expressive power availed to the user.

As we have seen, executive patterns basically determine the expressive power available at the PSM level, that is the (low-level) abstraction layer of the entities that populate the system at run time.

Determined patterns live entirely in the user-level design space; they are composed of primitive entities that can legally populate the PIM and therefore have scarce importance in the MDE process, since they do not augment the expressive power.

Declarative patterns instead arguably are the most important category of patterns in the context of an MDE process: they provide the user with a powerful abstraction that is automatically instantiated by model transformations and therefore fit perfectly into the model(s) product of the transformation.

It is then clear that declarative patterns are a crucial instrument for us to increase the expressive power we may avail to the user. Expressed graphically with reference to figure 2, we are increasing the slope of the lines that join the PSM space to the PIM.

One additional benefit of this classification is that makes it clear where the instantiation of each pattern occurs. Figure 3 provides a diagrammatic representation of this element of information: determined patterns clearly populate the *application layer*, since they are composed of legal entities of the user-level design space; executive patterns map or encode features of the execution environment, whether software (kernel or middleware or both) or hardware; declarative patterns, finally, reside at PIM level, but outside the direct projection of the PSM expressive power and thus require model transformation to come in existence as a correct by construction assembly of legal PSM entities and constructs.

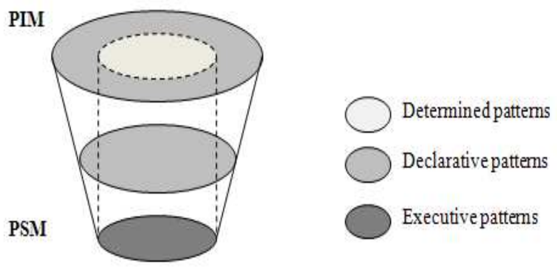


Fig. 3. Patterns in model-driven engineering. Determined and declarative patterns are avaliable in the user modeling space. Determined patterns result from direct projections of the PSM expressive power. Declarative patterns exist beyond the projection of the PSM expressive power and require model transformation to be implemented in terms of legal PSM entities. Executive patterns solely exist in the PSM level.

Let us now determine whether and to what extent the user should be aware of patterns, according to their class of belonging.

Determined patterns are obviously known to the designer, who is the primary and sole actor in their use. Conversely, executive patterns are intentionally hidden to the user. The situation is not that clear instead for declarative patterns. Should the designer be aware that the decoration of some model attributes triggers the activation of a declarative pattern? Additionally, should the designer explicitly require the activation of a pattern or else simply rely on the intelligent support of the framework to recognize the conditions for its activation? Two analogies may help us illustrate the differences between these two approaches.

One of the classical optimization performed by a compiler is to move the invariant part of a loop outside the loop itself. The average programmer need not be aware that this optimization is performed. Conversely, let's imagine a programming language that prescribes that remote operations be flagged with a *remote* keyword; in this case the programmer intentionally determines the generation of stubs, skeletons, and all other elements required to perform a remote invocation. At present we do not have a strong position on the *entire class* of declarative patterns, and we are inclined to believe that the issue should be evaluated on a case by case (i.e., per pattern) basis.

4 Pattern catalogue

This section briefly discusses examples of patterns from the categorization we have just proposed. The examples we have chosen reflect problems typical of the real-time application domain. We concentrate on declarative and executive patterns, because determined patterns should be well known to the reader.

4.1 Partitions and communication filters

While the notions of partition and communication filters are not new to software engineering, especially in the high-integrity domain (see for example the ARINC-653 standard [17]), they still have to find their place in a model-based development.

The use of logical and physical partitions permit to attribute software and hardware components to distinct levels of criticality so as to guarantee the required level of isolation in time, space and communication among them.

The notion of partition must be part of the metamodel itself, because the designer must be able to consciously allocate multiple executable entities within given partitions. In that manner, the entities included in a partition inherit the partition criticality and benefit from the partition-level isolation mechanisms. Neither the causing of criticality inheritance nor the modeling of isolation mechanisms, however, are to be explictly performed by the user: they can instead be easily realized by way of model transformation. The notion of *partition* does thus reflect a declarative pattern.

Communication filters are tightly related to partitions. When a lower-criticality partition establishes a communication link with a higher-criticality partition, the integrity of the exchanged messages should be subject to verification. The execution of the higher-criticality partition may in fact be affected by the computation required by a lower-criticality partition. To only permit allowable communications (operation requests or reports of results), filters are suitably interposed between communicating partitions to perform all of the necessary verification on the permissibility of the communication.

Filters are part of the metamodel. However they are not to be used explicitly by the user, who need not even be aware of their existence, but rather they are put into place by the tail end of the model transformation process. The notion of *filter* does thus reflect an executive pattern.

4.2 Callback

The purpose of the *Callback* pattern is to extend the PIM expressive power beyond the projection of the PSM modeling space. The pattern can be categorized as both an executive and declarative pattern. It does not require any specific action from the designer (as thus equates to an executive pattern), but its application has consequences which must be known to the user, as they impact the software architecture and the interpretation of analysis results.

Let us use a simple example to illustrate the Callback pattern, which we base on the classical producer-consumer archetype. Both the Producer and the Consumer have a single method, respectively `produce` and `consume` (cf. figure 4). The operation of `produce` consists in (a) producing an item, (b) passing it to the consumer and (c) adapting its own behavior according to the Consumer's feedback. The operation of `consume` consumes the item and returns its feedback via an `out` parameter, `Feedback`.

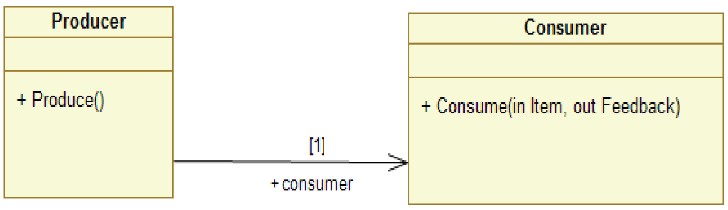


Fig. 4. Callback pattern: class diagram prior to the application of the pattern.

Following the HRT-UML/RCM modeling semantics for components, we declare the concurrent semantics on ports of provided services: the port providing `produce` is marked ≪cyclic≫, meaning that a dedicated task with a constant periodic release calls `produce`. The port providing `consume` is instead marked ≪sporadic≫, meaning that its invocation by a caller causes a sporadic task to be released to actually execute `consume` (cf. figure 5). Additional non-functional concerns, such as tasks priority and period or minimum inter-arrival time, are addressed by decorating the provided port of each component with specific attributes (which we omit in this discussion).

It is worth noticing at this point that the semantics expressed in this user model is *not* Ravenscar-compliant. The reason is that operation `consume` requires a synchronous communication (since its profile includes an `out` parameter), but the corresponding port is marked ≪sporadic≫, which makes it deferred in HRT-UML/RCM and thus necessarily asynchronous. The semantics intended by the user model is that of a synchronous deferred communication, which is forbidden in RCM. Interestingly however, a simple stage of model validation can notice the problem and trigger appropriate actions to resolve it.

To solve the problem and thus satisfy the user without incurring violations of the RCM restrictions, we first perform an *automated* model transformation on the class

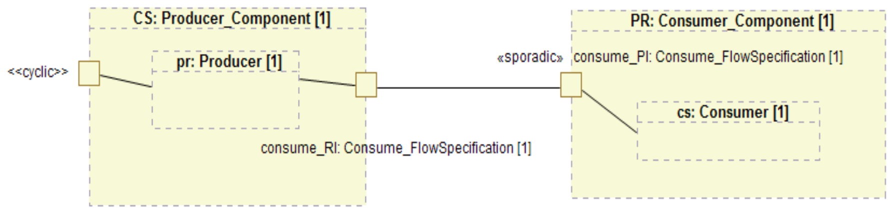

CS: Producer_Component [1]
pr: Producer [1]
<<cyclic>>
consume_RI: Consume_FlowSpecification [1]
PR: Consumer_Component [1]
«sporadic»
consume_PI: Consume_FlowSpecification [1]
cs: Consumer [1]

Fig. 5. Callback pattern: system model prior to the application of the pattern.

diagram (which is part of the so-called functional model in HRT-UML/RCM). The transformation changes the profile of the `consume` to accept a callback (i.e., a function pointer) instead of the `Feedback` out parameter. Its action semantics is then changed to: (a) declare a local variable in place of the missing `out` parameter; and (b) invoke the callback at the end of its execution, passing the above variable as its in parameter.

The action semantics of `produce` is then changed accordingly: in correspondence to the invocation to `produce` it is split into two separate methods. The action semantics coming *after* the invocation to `consume` is enclosed into `consume_callback`, which is the callback passed to `consume` (cf. figure 6).

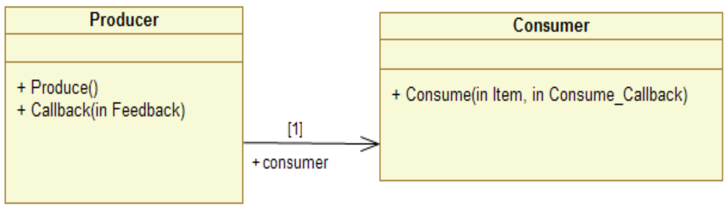


Fig. 6. Callback pattern: class diagram after application of the pattern.

The concurrent semantics declared in the component ports remains the same for those that provide `consume` and `produce`. The port providing `consume_callback` is instead marked ≪sporadic≫ so that a dedicated task may ensure a prompt response to the invocation of the callback from `consume`.

The resulting model is Ravenscar-compliant again as the services provided by the ports marked as ≪sporadic≫ do not include `out` parameters (cf. figure 7) anymore.

At this point an additional model transformation may generate the SAM and assign the user-specific functional behavior to it. Dedicated tasks are created for `produce` (cyclic), `consume_callback` (sporadic) and `consume` (sporadic) and the respective methods are allocated to the fully-legal main operation of the respective tasks. Message queues protected against concurrent access are created for `consume` and `consume_callback` as the means to implement asynchronous communication between the caller and the sporadic task on the side of the callee. A further shared re-

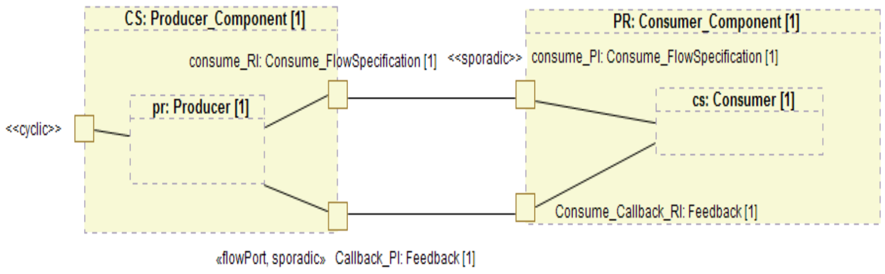

CS: Producer_Component [1]
consume_RI: Consume_FlowSpecification [1]
<<sporadic>>
PR: Consumer_Component [1]
consume_PI: Consume_FlowSpecification [1]
cs: Consumer [1]
pr: Producer [1]
<<cyclic>>
Consume_Callback_RI: Feedback [1]
«flowPort, sporadic» Callback_PI: Feedback [1]

Fig. 7. Callback pattern: system model after application of the pattern.

source may be automatically generated to safeguard the concurrent access the tasks behind `produce` and `consume_callback` to application data that the split of the single user-level operation should not duplicate.

Source code is finally generated from the transformed model and from the SAM; no code is instead generated from the original user model.

5 Discussion

By the introduction of the executive and declarative pattern categories, we achieve two results which are beyond the reach of classical determined patterns.

Executive patterns, like the Filter pattern, relieve the designer the need to specify complex yet recurrent parts of the software behavior, and rather introduce them directly in the PSM (and, possibly in the source code too).

Declarative patterns instead, serve two distinct purposes. First of all, they reduce the amount of modeling effort required for the designer to express the semantics needed to solve an application-level problem: this is for example the case of the Partition pattern.

Declarative patterns may also help extend the expressive power availed at PIM well beyond the perimeter permissible to the PSM: this is the case of the Callback pattern.

In fact, we see declarative patterns as a most promising pattern category for high-integrity systems in general and real-time systems in particular. They have the potential to release the user-level modeling process from (some of) the restraints that are propagated upwards from the underlying analysis theories, and thus extend the expressive power availed to the user. In practice the restraints that may be lifted are those for which declarative patterns exist which permit model transformations that provably preserve the properties of interest down to the final implementation.

Interestingly enough, declarative and executive patterns are both realized by means of property preserving model transformations and take the form of correct by construction aggregates of primities entities of the metamodel. While those patterns are not primitive entities themselves, they do exist in the metamodel space because it is their very existence under the semantic constraints enforced by the metamodel that asserts their legality. A key implication of this stipulation is that the compositional logic of the model transformations must be defined as an integral element of the metamodel itself.

What we currently see as the main limitation to the full application of declarative patterns is the impact they may have on the functional (and not only architectural) specification of the system. We have seen a glimpse of this problem in the application of the Callback pattern, which caused us to break a single functional operation into two distinct parts.

To date, mainstream modeling technologies have failed to provide a full and usable representation of action semantics in the metamodel space. This limitation constitutes a technological (though not conceptual) hurdle to our endeavor.

In several, perhaps most cases, declarative patterns cannot be silently applied by the modeling infrastructure as they may considerably increase the distance between the user space and the part of the PSM that is the product of automated model transformation. The larger the distance the more complex to understand the end results of the transformation, in both qualitative and quantitative terms. For example, the application of the Callback pattern not only modifies the functional specification provided by the user, but also impacts the concurrent and synchronization properties of the system by creating an additional task and an additional shared resource. The modeling infrastructure should thus justify all model transformations and provide evidence of traceability between levels of abstraction.

6 Conclusions

The production of formal analysis models is a complex task which requires to verify that (i) the designed model does not evade the boundaries of the semantics permitted by the underlying analysis theories; and (ii) the semantics is preserved at each abstraction level crossed by transformation.

The preferred way to satisfy both requirements is to constraint the modeling space by directly projecting the expressive power of the PSM onto the PIM modeling space (with the side benefit of easing the automated production of the PSM).

We introduced two new categories of patterns specific of the MDE context to add to the well-know category of determined patterns that stem from [14]. Executive and declarative patterns are meant to address other issues than those targeted by classical design patterns.

Executive patterns are expected to deliver the user from the need to specify parts of the architecture and functional behavior of the application by embedding consolidated solutions directly in the PSM (possibly the source code).

Declarative patterns instead extend the expressive power of the PIM by relaxing semantic constraints that hold on the PSM. The implementation of the semantics implied by declarative patterns is naturally realized by automated model transformation.

We discussed three specific instances of patterns in the above categories, which address problems recurrent the real-time systems domain.

The introduction of those new categories of patterns promises to increase the expressive power attainable at user level and an interesting new dimension of automation in the model-driven engineering of real-time systems. The full exploitation of those patterns requires a highly integrated modeling infrastructure, which includes a modeling language, and a set of proven and correct by construction model-to-model and model-to-code transformations.

References

1. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2) (2006) 25–31
2. SAE: Architecture Analysis and Design Language. `http://la.sei.cmu.edu/aadl/currentsite/aadlstd.html`.
3. OMG: UML profile for MARTE. (2007) `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`.
4. OMG: SysML specification. (2007) `http://www.omg.org/cgi-bin/doc?formal/2007-09-01`.
5. Bordin, M., Vardanega, T.: Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In: Reliable Software Technologies - Ada-Europe. (2007)
6. Panunzio, M., Vardanega, T.: A Metamodel-driven Process Featuring Advanced Model-based Timing Analysis. In: Reliable Software Technologies - Ada-Europe. (2007)
7. ASSERT: `www.assert-project.net` (2004-7)
8. Chapman, R.: Correctness by Construction: a Manifesto for High Integrity Software. In: ACM International Conference Proceeding Series; Vol. 162. (2006)
9. Vardanega, T.: A Property-Preserving Reuse-Geared Approach to Model-Driven Development. In: 12th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications. (2006)
10. Bordin, M., Panunzio, M., Vardanega, T.: Fitting Schedulability Analysis Theory into Model-Driven Engineering. In: Proc. of the 20th Euromicro Conference on Real-Time Systems. (2008)
11. Pulido, J.A., de la Puente, J.A., Hugues, J., Bordin, M., Vardanega, T.: Ada 2005 Code Patterns for Metamodel-based Code Generation. Ada Letters **XXVII**(2) (2007)
12. Zalewski, J.: Real-time Software Design Patterns. In: 9th Conference on Real-Time Systems. (2002)
13. Sanz, R., Zalewski, J.: Pattern-based Control Systems Engineering. IEEE Control Systems Magazine **23**(3) (2003) 43–60
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison - Wesley (1995)
15. Pratt, T.W.: Programming Languages. Design and Implementation (Second Edition). Prentice-Hall (1984)
16. Burns, A., Dobbing, B., Vardanega, T.: Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. Technical Report YCS-2003-348, University of York (2003)
17. ARINC: Avionics Application Software Standard Interface: ARINC Specification 653-1 (2003)